AN INVESTIGATION INTO PARTITIONING ALGORITHMS FOR AUTOMATIC HETEROGENEOUS COMPILERS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Antonio Michael Leija

September 2015

COMMITTEE MEMBERSHIP

TITLE:	An Investigation into Partitioning Algorithms for Automatic Heterogeneous Compilers
AUTHOR:	Antonio Michael Leija
DATE SUBMITTED:	September 2015
COMMITTEE CHAIR:	Dr. Andrew Danowitz, Ph.D. Assistant Professor of Electrical Engineering
COMMITTEE MEMBER:	Dr. John Oliver, Ph.D. Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Dr. Bridget Benson, Ph.D. Assistant Professor of Electrical Engineering

ABSTRACT

An Investigation into Partitioning Algorithms for Automatic Heterogeneous Compilers

Antonio Michael Leija

Automatic Heterogeneous Compilers allows blended hardware-software solutions to be explored without the cost of a full-fledged design team, but limited research exists on current partitioning algorithms responsible for separating hardware and software. The purpose of this thesis is to implement various partitioning algorithms onto the same automatic heterogeneous compiler platform to create an apples to apples comparison for AHC partitioning algorithms. Both estimated outcomes and actual outcomes for the solutions generated are studied and scored. The platform used to implement the algorithms is Cal Poly's own Twill compiler, created by Doug Gallatin last year. Twill's original partitioning algorithm is chosen along with two other partitioning algorithms: Tabu Search + Simulated Annealing (TSSA) and Genetic Search (GS). These algorithms are implemented inside Twill and test bench input code from the CHStone HLS Benchmark tests is used as stimulus. Along with the algorithms cost models, one key attribute of interest is queue counts generated, as the more cuts between hardware and software requires queues to pass the data between partition crossings. These high communication costs can end up damaging the heterogeneous solution's performance. The Genetic, TSSA, and Twill's original partitioning algorithm are all scored against each other's cost models as well, combining the fitness and performance cost models with queue counts to evaluate each partitioning algorithm. The solutions generated by TSSA are rated as better by both the cost model for the TSSA algorithm and the cost model for the Genetic algorithm while producing low queue counts.

ACKNOWLEDGMENTS

Through the five years I’ve spent at Cal Poly I have countless people to thank. Firstly, I want to thank Dr. Danowitz for taking me under as one of his first graduate students and Dr. Oliver for pointing me towards the master’s degree. Also I would like to thank Toby Elder, Ross Johnson, Dan Rodman, Colin Mitchell, Cody Smith, Bryn Thompson, and Cynthia Wong: thank you for the dank adventures, ninja tag, and awesome bike rides. To the CPE Networks Crew: Michael Peterson, Elmer Urbano, Anthony Cisneros, Shaun Boley, Efren Cabebe, and Cale Glisson, for all of the late night coding sprints, dollar tacos, and Tibetan throat singing. To all of the PolyCon members, art night attendees, and everyone else who kept me sane with non-school related activities. To Troy Weipert and the rest of the ANTS group, for making my part-time work experiences the best any Poly student could ask for. Finally I would like to thank my mother, father, and two brothers for being there for me, and being constantly supportive through this endeavor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

1 Introduction

This thesis will investigate partitioning algorithms for automatic heterogeneous compilers in regards to their design and performance, establishing an apples to apples comparison of these partitioning algorithms. Firstly, the background and previous work in the field of AHCs will be examined, and algorithms will be selected for implementation and comparison in a real AHC: Cal Poly's own "Twill" [1] compiler. After that the results of the algorithms will be examined.

Heterogeneous approaches to the design of computer systems has become a popular approach for high throughput data processing. Accelerating programs to potentially 40x of their normal speed is possible by taking advantage of heterogeneous parallelization. [2] Due to the silicon wall [3] with regards to transistor sizing [4], offloading [5] calculations onto heterogeneous hardware has also become an effective solution. Heterogeneous approaches are usually composed of a traditional processor combined with dedicated hardware logic. This can come in the form of a Field Programmable Gate Array (FPGA) [6] combined with a processor. This enables a common software interface [7], such as a command line interface (CLI) or graphical user interface (GUI), to combine [8] with accelerating hardware such as an FPGA. Interfaces exist between the two domains, usually in the form of queues or shared memory. In a heterogeneous design, software brings with it the expense of software engineers and test engineers. Adding hardware means that verification engineers become a requirement along with RTL engineers. This complexity results in higher design costs, specialized engineers, and longer project times.

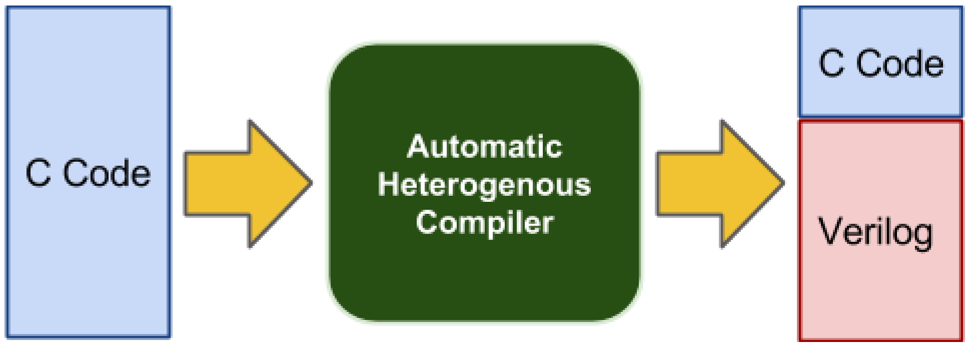


Figure 1 Illustration of an Automatic Heterogeneous Compiler

Automatic heterogeneous compilers are fairly simple in their operation, as shown in Figure 1. Ideally they simply take some software code, and output a blended solution of software and hardware, such as C and Verilog. AHCs empower design teams to easily create a heterogeneous design without the trappings of a dedicated digital design team. Already preliminary technologies have attempted to explore the area of heterogeneous automated design such as CHiMPS [9], GreenDroid [10], and Cal Poly's own Twill [1].

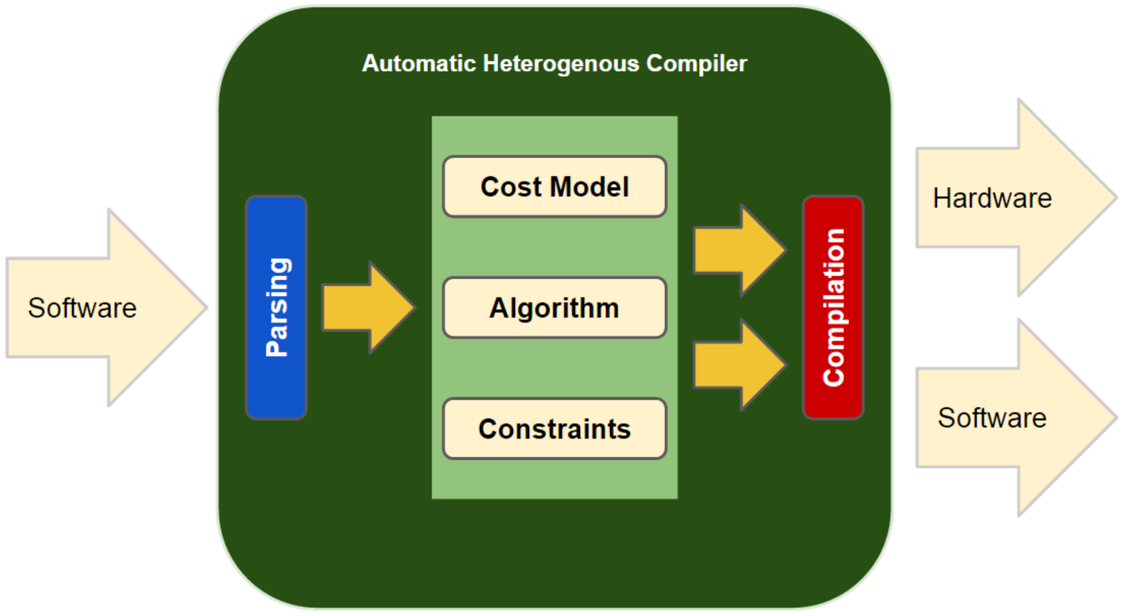


Figure 2 Components of an Automatic Heterogeneous Compiler

These tools [8] use parsers [11] (shown in Figure 2) to read in a given software language [12], followed by partitioning algorithms to choose which pieces of the original software go into a hardware partition. After this a compiler (or compilers) automatically create the necessary interfaces between hardware/software crossover points and convert the partitions into software code and a hardware description language (HDL).

In Figure 2 it can be seen that the core of the AHC is made up of three components: cost models, algorithms, and constraints. Partitioning algorithms for heterogeneous computing have already been devised in order to find automatic solutions that prioritize the speed of the finished solution, time taken to find the solution, area usage, power usage, or other metrics. The algorithm designs include genetic selection, knapsack problems, and simulated annealing. However these algorithms have not been formally tested against one another.

Partitioning algorithms also require a representation of the input software, and a cost model. The representation [13] can be done in many ways but usually it is assumed to be a graph of collected instructions (called basic blocks) separated by branches. Cost models are used to estimate the effects of putting a piece of the original software into hardware or software. These effects [14] can come in the form of latency, area usage, or power usage. Cost models also predict the outcome of a whole solution, and attempt to constrain issues such as having a limited FPGA area, or a maximum time constraint. Without these kinds of estimations, constraints, and predictions, the partitioning algorithms would lack necessary data to create an optimal solution for an heterogeneous system.

This paper will implement and investigate three partitioning algorithms for AHCs: Twill's original accumulator solution, Simulated Annealing + Tabu Search, and Genetic Search. While investigating these algorithms, novel results about the suitability and performance of these three algorithms will be drawn from the data gathered inside Twill.

This paper will be organized as such: Section 2 will give a background in the work done in the area of automated heterogeneous design. Section 3 will present an overview of the terminology that will be used. Section 4 will go into the design of the algorithms chosen. Section 5 will examine the way metrics will be collected and analyzed. Section 6 will cover the actual implementation of the algorithm and data collection. Section 7 will summarize the collected data and present future work.

2 Previous Work

This chapter will examine space of automatic heterogeneous compilers, partitioning algorithms, and cost models. Current automatic heterogeneous compilers such as CHiMPS and Cal Poly's very own Twill compiler are covered along with a handful of heterogeneous partitioning algorithms and current work in cost models. Chapter 3 will cover definitions that will be used when implementing the algorithms and in Chapter 4 the chosen algorithms from this chapter will be covered in further detail.

2.1 Heterogeneous Frameworks

Designing a blended hardware software solution from the ground up can be difficult and expensive, but many frameworks [15] have been proposed to alleviate some of the necessary design requirements for a heterogeneous system such as a hardware-software communication designs, block standards for the off-CPU hardware, and methodologies [16]. This sub section will review the current work in the area of heterogeneous compilers.

2.1.1 GreenDroid

Developed by Goulding et. al. GreenDroid is a multi-core prototype with a focus on smaller cores called conservation cores (c cores). [10] A host Central Processor Unit (CPU) uses these c-cores to carry out the frequent repetitive tasks found in programs. Each collection of c-cores can be considered as a tile in the GreenDroid design, and each tile has a defined 32 Kbyte L1 cache which is coupled to the host CPU. This 32 Kbyte cache is considered to be a shared cache, meaning that multiple devices can use the cache simultaneously. This kind of hard definition helps the software and hardware interface on a common ground.

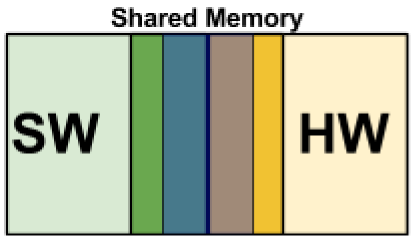


Figure 3 Hardware and Software Interfaces (Shared Memory)

This common ground being a simple format: shared addressable memory, which is shown in Figure 3. This constraint means that designers are alleviated of having to design a hardware/software interface, and that when the interface is worked on it is well defined. On top of having this constrained communication framework in GreenDroid, the c-cores themselves are tightly defined with a seven stage pipeline, a single floating point unit, 16 Kbytes of instruction cache, translation lookaside buffer, and the 32 Kbyte cache mentioned before. This means that the hardware portion is already defined as well, unlike a custom FPGA design that may have wildly unnatural interfaces and designs. Software designers need only to create a blend of host CPU software and c-core software to take advantage of the GreenDroid heterogeneous framework.

This kind of restrictive framework means that implementation is standard for the c-cores, and that software designers can easily start using the technology since there will be no RTL necessary. Also since the cores are pre-defined, interaction between the c-cores and the host CPU will already have management hardware or software in between to manage data sharing between the various threads of the whole system. GreenDroid is limited by the c-cores however, and this system can be considered to be "limited heterogeneous" because of the well-defined c-cores.

2.1.2 CHiMPS

Standing for Compiling High-level languages into Massively Pipelined Systems, CHiMPS is a high level synthesis tool that is intended to be used on a CPU and FPGA. It takes C code and turn it into CHiMPS Target Language (CTL) instruction blocks. These blocks can then be processed into HDL or reversed back into C, allowing a heterogeneous solution to be created. The concept of taking a common programming language and generating a blended solution that can run on a normal CPU core and custom logic is more flexible than GreenDroid in its design, but loses the elegance of a shared cache and constrained design methodologies like the c-core. This flexibility of custom logic eliminates unneeded parts of the c-cores or may allow operations that the c-cores could still not do effectively. CHiMPS was able to be tuned using #pragma statements that could throttle cache updates, implementation styles, loop unrolling, and other features. Much like compiler passes, these controls allowed some customization in the output generated by CHiMPS. Speedups compared to original C code were present but quite varied from 2.1x to 36.9x with a mean of 6.7x. At most 428.5% of the available hardware space was taken up, and at least 4.4% of the area was taken up. Keeping automatic designs within constraints of an FPGA in areas such as LUT counts, which is a physical limitation to the chip, is difficult and the use of an algorithm or threshold to eliminate impossible solutions is required, and CHiMPS had issues managing the available hardware properly.

2.2 Partitioning Techniques

In the field of hardware software partitioning, a large amount of prior work has been done in devising new heuristics. These algorithms attempt to tackle the NP-hard problem of hardware software partitioning with a variety of “natural phenomena” such as imitating evolution, annealing, or backpack packing. [17] Cost models will estimate the effects of placing an instruction in hardware or software, thus aiding the algorithm in rapid simulation. The estimations include data on an instructions implemented area, power, or latency cost. In testing they also tend to not use real input code, and instead settle on a Gaussian distribution of arbitrary cost values to represent a program dependence graph. True implementation of the algorithms is not common and many solutions end up not being fully realized.

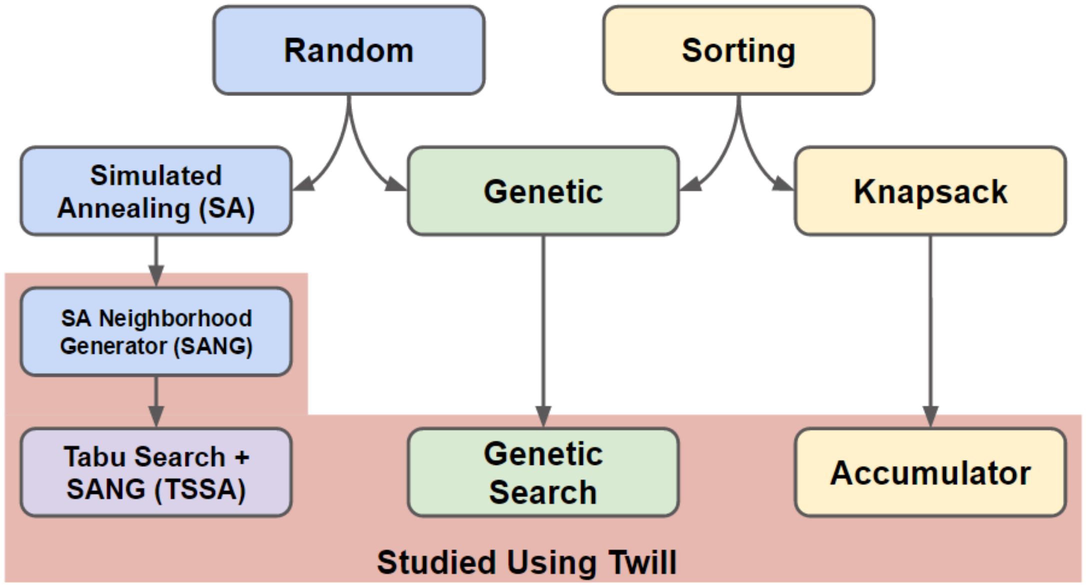


Figure 4 Taxonomy of Algorithm Approaches

As seen in Figure 4, there are two major traits that partitioning algorithms pull from: randomness and sorting. These two major traits play roles in three styles of algorithms, which in turn create four actual algorithms that we use. The styles of algorithms will be covered in the following sections, giving an overview into Simulated Annealing, Tabu Search, Genetic, and Knapsack algorithm families.

2.2.1 Simulated Annealing

Inspired by annealing metal [18], variables in the system include temperatures and cooling rates. Randomness and greedy selection is employed in simulated annealing. Some initial starting point (in the case of HW/SW heterogeneous design: a pure software solution) for the solution is used along with a starting temperature. The temperature goes into an pseudorandomm exponential equation that dictates the change of a given piece of the solution.

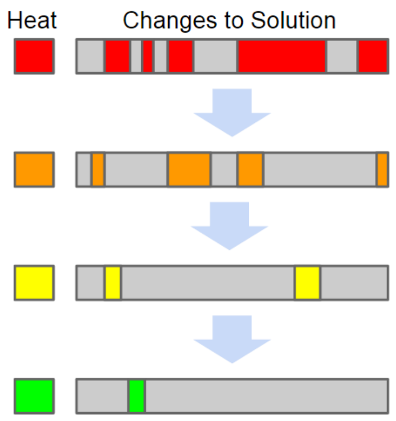


Figure 5 Simulated Annealing Visual of Heat Chaning a Solution

Imagine that Figure 5 consists of a bar showing encountered instructions and the changes made to the HW/SW solution colored in with the heat color, along with an indicator of the heat of the solution on the left. With a higher temperature, more random changes are prone to happen. As the temperature is decreased, further iterations will result in fewer changes until the solution is “cooled” resulting in no further changes. As the solution cools down, a local search is conducted near solutions that are close to the current solution. Nearby better solutions will become the current solution, and the process of adding randomness and cooling further will continue. This cooled solution should ideally be the global minima or maxima [19] in the field of possible solutions it is however prone to issues such as being stuck in valleys caused by local minima. This is due to the fact that this algorithm design is greedy by nature. It does not hold memory of previous runs, and does not attempt to follow a heuristic. However the gradually "cooling" randomness factor added is intended to move around the solution enough at high energy levels to trump becoming stuck in a local minima/maxima as most basic greedy solutions do.

In the paper "Integrated Heuristic for Hardware/Software Co-design on Reconfigurable" Devices by Liu et. al. a hybrid algorithm using Simulated Annealing (SA) and Tabu Search (TS) is explored. The SA portion will be reviewed here and the TS portion will be explored after. The SA portion is a constructive partitioning, meant to achieve a solution given some constraints. Wang et. al. uses SA to generate a small local set of results, and to hone these results using TS. To evaluate performance of an SA solution, a few cost metrics are used. Predecessor instruction latencies for a selected task along with successor instruction latencies are included, along with the latency tradeoff of a given task in hardware or software. The communication penalty between one node and another is also considered, along with the total penalty of a note (with its predecessors) to another node. The total performance of a system can be found by evaluating a simple equation that takes the communication penalties into account. Using this final performance calculation, a given simulated annealing solution can be evaluated against another.

2.2.2 Tabu Search

Tabu search is a search created by Glover in 1986, and is a metaheuristic [20] (or in our case, an iterative partitioning pass) that focuses on neighboring solutions. By examining partitioning solutions nearby to a given partitioning solution, an exhaustive graph of solutions can be explored greedily without having to generate the whole graph. This allows an extensive exploration while having consistent memory/processing requirements throughout the runtime, since the Tabu Search is a localized search. At any time there is only the solution under investigation, it's local neighboring solutions, and the best solution found so far. Each solution is considered in a graph of possible solutions.

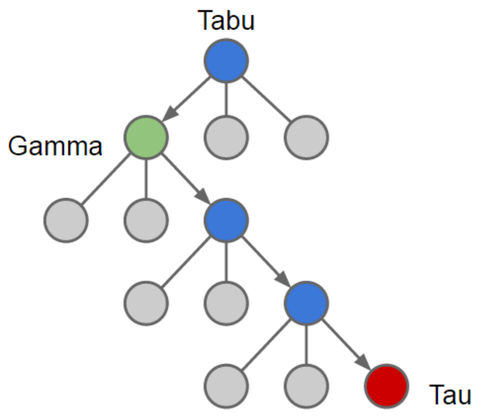


Figure 6 Example of a Tabu Search across a Graph of Solutions

There are 3 kinds of solutions: gamma solutions, tau solutions, and tabu solutions. Tau solutions are the winning solutions of local searches, which are compared against gamma solutions (the best-found-so-far solutions). Upon being "better" than the gamma solutions, a tau solution becomes the next gamma solution. Regardless of this result, the tau solutions become tabu solutions. Tabu search hinges on tabu solutions, which attempt to solve the local minima/maxima problem by enforcing a rule that examined solutions go into a tabu list. Any solutions considered to be "similar" to tabu solutions are avoided for some number of iterations of the tabu search. Similarity of solutions is user definable, so ranges of solutions can be considered similar. Tabu solutions may be bad or even good solutions choices, resulting in a "worse" local search overall, but they attempt to force the search to look in areas that a typical greedy search would avoid.

Wang et. al. uses the Tabu search to refine their SA pass, Tabu Search can be used to refine any pass such as genetic, or even uniform randomness.

The tabu search does not have anything that defines it as a HW/SW partitioning method, because at this point the solution is abstracted away into a singular "performance" score.

2.2.3 Genetic

Genetic algorithms [21] use the idea of natural selection to drive solution formulation, with a population of solutions having a "genome" complete with alleles that determine characteristics of each member of a generation. The generation is examined to find the best performing solution by comparing fitness ratings. After this, two high fitness solutions are randomly chosen, and their genomes undergo crossover with one another. Then these new genomes undergo mutation, resulting in children for the next generation. This is done until a new generation is at a sufficient size, and then the process starts over once again. The best solution across all generations is considered to be the best solution. This relies on mutations and crossover to generate enough variance in the population to avoid getting trapped in a false "best case" found inside a local minima or maxima.

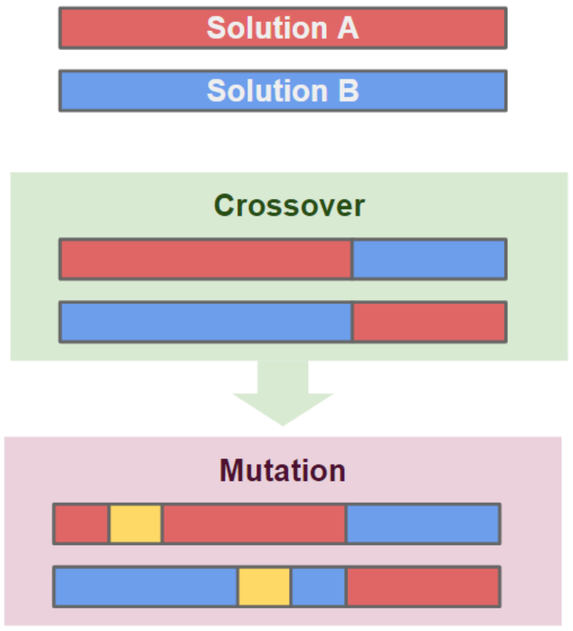


Figure 7 Visualization of the Genetic Method with Two Genomes

Mishra et. al. [22] proposes applying this theory of genetic selection into the space of hardware/software partitioning. They begin their algorithm with a population that has a genome, which is a bit pattern defining which portions of the original software solution go into hardware or software. After this a scoring must be given to the solution. The Objective Function (OF) is responsible for calculating the fitness of a solution, which is the scoring mechanism for the Genetic algorithm. For this implementation of a genetic algorithm, the communication costs between HW and SW edges are not considered. This can result in numerous SW/HW jumps.

2.2.4 0-1 Knapsack

A knapsack solution [17] is quite simple, where each piece of a solution is visualized as a box, and a knapsack exists that must be filled with the boxes. Ordered by priority according to a given cost model, the pieces of the solution are stored into the knapsack until there is no more room. This result is akin to taking the best pieces of a solution along, emphasizing a limited amount of space/resources represented by the size of the knapsack. [23]

In the research done by Chen et. al: "One-dimensional Search Algorithms for Hardware/Software Partitioning", the NP-hard problem of HW/SW partitioning is attempted to be surmounted using a one dimensional 0-1 knapsack search. Giving a knapsack capacity of K, and a set of items S, they attempt to find a subset to maximize their profit (score). In order to greedily fill the knapsack, the profit to weight ratios is ordered so that the most lucrative options are "packed" first.

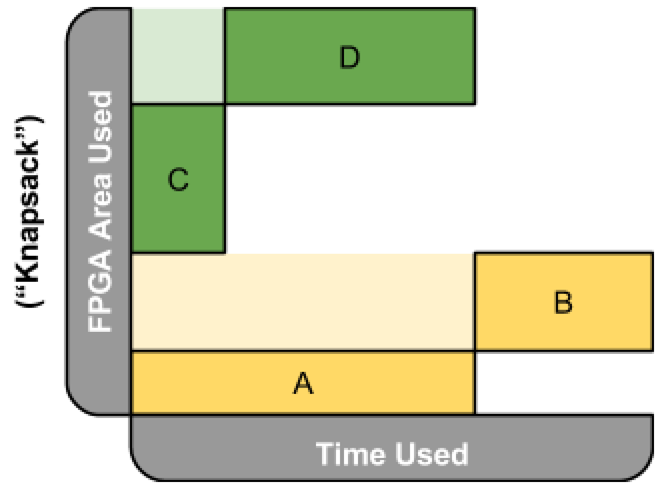


Figure 8 Parallel Tasks on Limited Discrete Hardware

As seen in this illustration, FPGA area is a limited resource, but time is not as limited. Given a program that has blocks A, B, C, and D where B is dependent on D and A, and D is dependent on C, we desire to implement the components in hardware. Observing the FPGA dimension purely, we can see no overlap between A, B, C, and D since they must fill in distinct areas of the FPGA. However, since the resource of time can be run in parallel, we run C+D simultaneously with A.

Chen et. al. describes the Software and Hardware cost to be the following, with x being a solution of problem set P. SW and HW costs are defined by a scalar constant. The communication cost between nodes is also defined by an arbitrary scalar cost to represent a queue between hardware and software elements. These costs can be used to visualize the "ideal" minimization / maximization problem P and Q.

2.3 Cost Models

In order to dictate how well (or poorly) instructions in a task graph may perform as hardware or software, cost models [24] are required to predict the outcome using traits like cycle time, area usage, and power usage. These models [25] may have one, some, or all of the following traits noted before, and algorithms can be tuned to focus on a subset [26] of the traits included in the model [27]. This allows automated design that can be aware of latency and area usage.

The scoring of cost factors such as cycle time, area, power, and communication costs are evaluated as a unitless number. Many models use a scalar value for software cost of that particular node. What is this cost? It is not described, or calculated. It just "is". Solutions can still be reached this way, but it helps to have some more realistic cost models in order to correctly estimate the outcome of the partitioning algorithm. [28]

Work done in this area, such as Rupp et. al. in "Static Estimation of Execution Times for Hardware Accelerators in System-on-Chips" [29] predict worst case execution time and best case execution time with a control flow graph representing various operations to evaluate. Fidelity calculations are used to find Worse/Best Case Estimated Time results, and can be used to find an execution time profile. Since hardware offloading allows speed increases, the latency of instructions in hardware and software are extremely important.

2.3.1 Cycle Time

Called cycle time, or latency, this is considered to be how long it takes for a given set of instructions to finish. In software, it is how many clock cycles will be necessary to complete the instruction, while in hardware, it is how many clock cycles will pass until the hardware has completed its “instructions”. Traditionally hardware is faster than software in this cost domain. Hardware also has the ability to run in parallel, since logic evaluation can happen instantaneously in its own dedicated area of silicon disregarding a normal pipeline. This opens up the prospect of parallelization, since multiple instructions in silicon can run at the same time, as opposed to a typical pipeline. One caveat is that simultaneous hardware implementations are still limited by the longest instruction, queuing logic, and variable dependencies.

2.3.2 Area

Area is one cost trait that software trumps when compared to hardware. For software, all instructions share the same area: being the silicon processor the software is running on. For hardware: each instruction/logic network has to have it’s own dedicated physical area. This is a major constraint in design, as an FPGA is not limitless. Physical area can rapidly get very expensive with the use of digital signal processing (DSP) blocks, or with massive parallelized operations. Traditionally this trait is inversely related to the cycle time benefits, as repeated operations or sets of similar operations tend to be parallelizable. One excellent example is a for-loop or matrix.

2.3.3 Power

Power is a blended trait [30], as the power use efficiencies of software or hardware can be lucrative depending on the application. If a processor runs too long, it will burn a small amount of power over a long period of time, whereas hardware, while being fast, can burn a large amount of power in a short amount of time. Power usage can be tuned in software by clocking down a processor, or by using dark logic controls in hardware to turn off unused logic areas at certain times in a program.

2.3.4 Crossings

In the space of heterogeneous solutions, the extra cost of interfaces between hardware and software is a critical concern, as each crossing adds time, power, and area to a design that did not have it beforehand. As such, extensive cuts in a task graph may result in the area costs of crossings becoming more expensive than the actual benefits found by making the cuts in the first place. This tradeoff is one that must be considered and managed in automated design.

2.4 Twill

Doug Gallatin's hardware software cosynthesis tool chain called "Twill" is designed to take C code and creates a blended solution using a hard-coded partitioning heuristic and cost model. Twill's tool chain creates fully functional logic and software, running the software off of a soft-core processor. Twill contains the possibility to have its heuristic and cost model modified. This platform lets the algorithms in question for heterogeneous solutions be tested in a live environment with real instructions to process.

2.4.1 Inspiration

Twill draws from earlier heterogeneous compilers, such as the NAPA C compiler [31], CHiMPS, and ROCCC systems. These all intended to take care of loops by parallelizing them in an FPGA, and in some cases solve difficult issues such as recursion. Overall these systems took software code in and split the code into hardware and software.

As heterogeneous systems started to become more popular, the need for operating systems to be designed with off-processor logic grew. Projects such as ReconOS [32], hThreads, and other RTOS/OS centric heterogeneous support frameworks were created to support the output of heterogeneous compilers. These are thread based to allow multiple processes to run. However, these threads are designed to exist in two possible states: hardware and software. This sort of awareness makes projects such as hThreads lucrative to heterogeneous combinations of software and hardware because since they are well defined, they impose some constraints on how processes interact.

Systems such as SPARK [33] and LegUp [34] have attempted to put the elusive hardware design aspect in the hands of the software programmer as opposed to the hardware engineer by allowing users to rapidly deploy hardware solutions. This means that all a software engineer is required to do is write C code with minimal knowledge of hardware design. Then the C code can be readily converted into synthesizable RTL. This was also a desired trait for Twill, since writing a heterogeneous system can be complex.

Twill sought to unify the heterogeneous compiler, HW/SW interface framework insulation, and software engineer empowerment into one toolchain flow.

2.4.2 Architecture

Twill was created with three major parts: a compiler, a software runtime, and a hardware runtime.

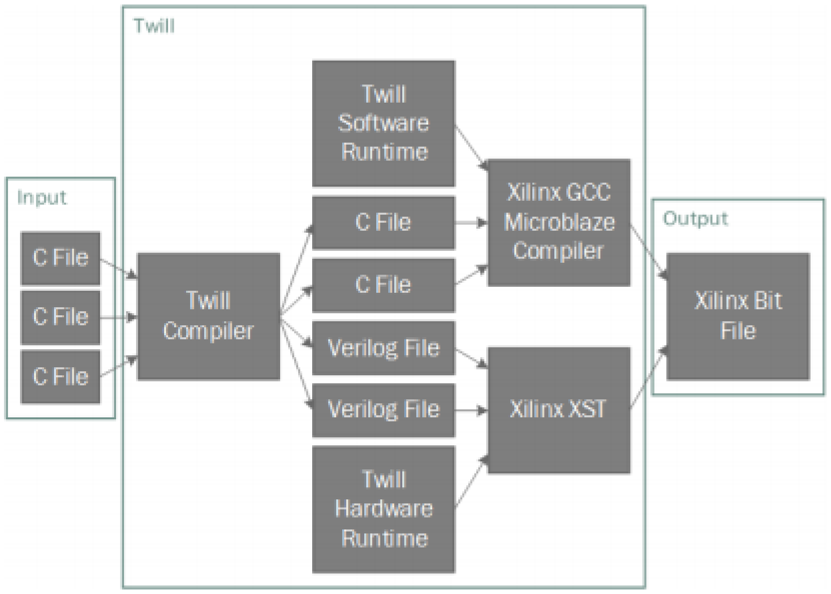


Figure 9 Twill Architecture with Input and Output Files [1]

The Twill compiler takes a single threaded C program and outputs a variety of C and Verilog files. The Verilog files are combined with the Twill hardware runtime and be synthesized using Xilinx's XST. The C files are combined with Twill's software runtime and undergo a final compilation pass in Xilinx's GCC Microblaze Compiler. The result is a soft core processor running C code that interfaces with the hardware around it, which is defined by the Verilog. This is placed onto an FPGA.

2.4.3 PHI Nodes and Fake Dependencies

When examining software for potential heterogeneity, the order of execution does not matter. Only variables dependent on previous operations or calculations do. These dependencies are the "true" dependencies, while the original branching code flow is considered to be made up of "fake" dependencies. The start and the end of the function will be preserved, but past that the software that calls a heterogeneous function expects to put data in and receive data back. This insulation means that tearing apart the "fake" dependency code flow is fine as long as the "true" dependencies are maintained so that the function can still return a proper value.

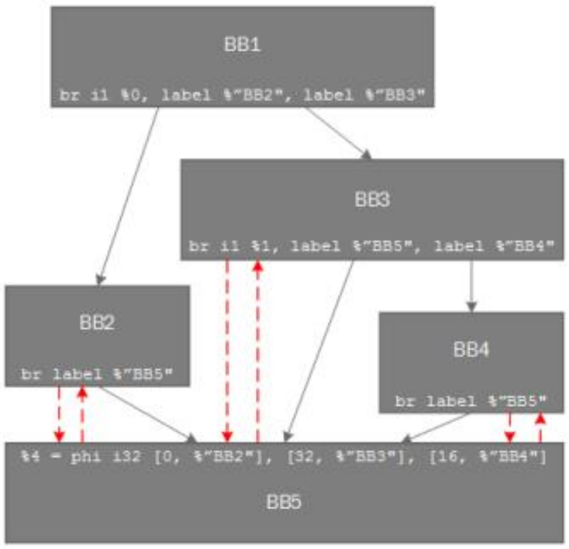

BB1
br i1 %0, label %"BB2", label %"BB3"
BB3
br i1 %1, label %"BB5", label %"BB4"
BB2
br label %"BB5"
BB4
br label %"BB5"
%4 = phi i32 [0, %"BB2"], [32, %"BB3"], [16, %"BB4"]
BB5

Figure 10 Basic Blocks with Fake Dependencies (Gray Arrows) and True Dependencies (Red Arrows) [1]

To visualize how the dependencies play into the code, let's examine the idea of basic blocks and PHI nodes in Figure 10. As shown, true dependencies can be found between BB3 and BB5, but not between BB1 and BB3. This means that BB3 and BB2 can process before or during BB1, however BB5 must wait for BB2 or (BB3 and BB4).

2.4.4 Twill Compiler

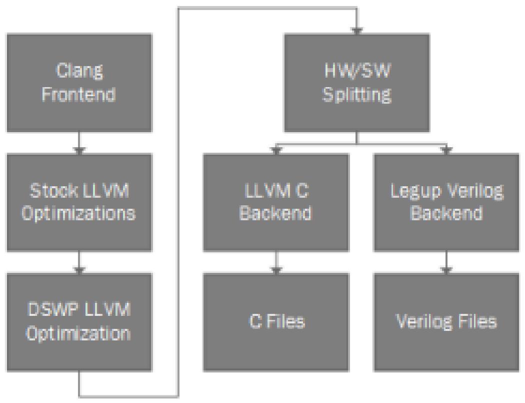


Figure 11 Twill Compiler Toolchain Flow [1]

The compiler relies on a few major pieces of software to run. Currently it is executed using a Python script, which in turn calls Clang, LLVM Transforms [35], and LegUp. [34] Clang (Figure 8) is responsible for turning the input C code into LLVM Instructional Representation (IR), so that the LLVM Transforms can operate on the exposed instructions. Standard and custom LLVM Transform passes are run in order to partition the code, garnering multiple files. LegUp is tasked with taking the hardware partitions and turning those into Verilog.

Clang is called with "-O2","-ffreestanding", and "-fno-builtin" flags to avoid LLVM manipulations to the memory that are not explicit in the original C. The point of Clang is to get the code into a workable IR, optimization is not a primary concern.

After Clang has run, the LLVM IR is now exposed. The following passes are run in order to prepare the IR for the custom Decoupled Software Pipelining (DSWP) pass: "basicaa", "mem2reg", "mergereturn", "lowerswitch", "indvars", "inline", "always-inline", "simplifycfg", "gvn", "adce", and "loop-simplify".

The custom DSWP pass ensures that both hardware and software will have address references to the global variables. A few more stock passes are run in order to prepare the IR for the Program Dependence Graph (PDG). These passes are “deadargelim”, “argpromotion”, and “constprop”.

The PDG is a graph that shows collections of instructions and their related instructions in the form of parents and children. These are control flow dependencies, and also “invisible” PHI node dependencies. PHI nodes show up where data is intended to be used, like a variable, but a block beforehand must calculate it.

The PDG is reliant on LLVM’s normal “basicaa” and “loops” information. Using this information, a graph is created with nodes containing a set of instructions. The loop data helps expose possible parallelism points. The nodes also will have a cost associated with them, and in Twill’s case it is the estimated cycles the instruction is expected to take.

Once this PDG has been generated, the DSWP pass runs a very basic partitioning algorithm to divide all the nodes of the PDG between the available partitions. The developer specifies how much of the program will become software as opposed to hardware.

In order to divide the nodes between the available partitions, a sort is conducted that finds all the PDG nodes that are able to be placed into the hardware partition. As the hardware partition fills up there is a check against the defined percentage. Once the partition has been “filled” the rest of the PDG is either placed into the next hardware partitions, or into software.

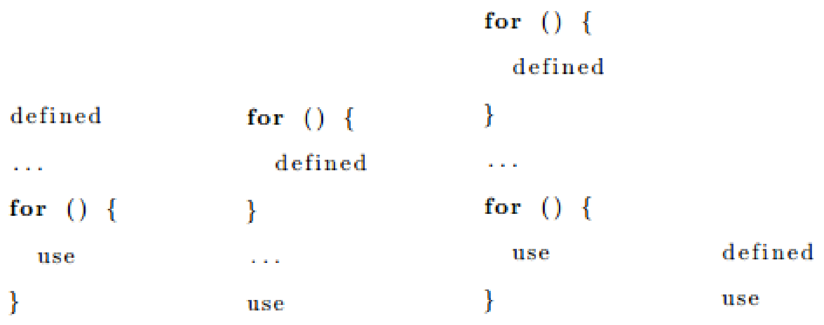
For nodes to be part of this partitioning process they must be able to be in either software or hardware. Nodes that may not fall into this category include the start of a function or the end of a function, or system/library calls such as printf().

After the division of nodes, enqueue and dequeue pairs are created to bridge crossings between hardware and software. This process establishes the new control flow to protect the PHI nodes. Then the resulting instructions for partitions are combined into basic blocks, only lacking branch and call site instructions.

Branches are added appropriate to branch targets, and PHI node dependencies are attempted to be resolved to avoid accessing data before it is available.

2.4.5 Control Dependencies

Twill attempts to find loops in programs, defined by for () blocks, and places the enqueue/dequeue pairs in four different cases of for block construction.



```
defined
...
for () {
  use
}
```

(a) use in a sub-loop of defined

```
for () {
  defined
}
...
use
```

(b) define in a sub-loop of use

```
for () {
  defined
}
...
for () {
  use
}
```

(c) define and use in distinct loops

```
defined
use
```

(d) define and use in same loop

Figure 12 Twill For Loop Pairs [1]

Identification of these loops are paramount, as repeated operations may not have dependencies on the previous operation (the 100th iteration of the loop needs no knowledge of any other iteration to complete). This makes certain loops a lucrative target for parallelization, leading to hardware designed to replace the for loop, doing up to 100 operations simultaneously. Repeated operations in code can come in a variety of ways, and can either be dependent or independent.

2.4.6 Hardware Software Splitting

Once the partitions are finalized, the hardware designated IR is sent off towards LegUp to be turned into .v sourcecode.

2.4.7 Future Work

A variety of hardware software partitioning heuristics can be chosen with similar traits or designs, along with a real cost model to be used inside the heuristics. With these heuristics and the modified Twill system it is possible to gather meaningful data and preform heuristic comparisons. Implementation and testing of these heuristics will give us the truth to whether or not they actually get results. Some may perform better than others due to their different approaches.

3 Definitions

In this section, important definitions used throughout this thesis are covered. A singular instruction, graph of instructions, or partition is carefully defined in order to make the implementation, testing, and results of this thesis clear.

3.1 General

SCC - Strongly Connected Component, a representation of an Instruction

Heterogeneous Solution - A program that is divided into hardware and software

Hardware/Software Partition - The portion of a heterogeneous solution in hardware/software

Program Dependence Graph - The graph of SCCs that is generated and used to make the software and hardware partitions

SCC Instruction Node - Also called an SCC, these contain the instruction and pointers to the next instruction.

Directed Acyclic Graph - A graph of instructions that exists initially as a software only homogenous solution

Partitioning Algorithm - An algorithm that splits up a homogenous software solution into a heterogeneous solution. Made up of a heuristic and a cost model

Solution Node - A singular heterogeneous solution intended to be used in a search graph, contains extra metadata describing the solution enclosed within

Heuristic - A function used to move across a graph of solution nodes in a manner dictated by a cost model

Cost Model – Used to generate a rating, dictates how instructions in a given solution are evaluated according to implementation costs in hardware or software.

4 Algorithm Overview

In this section the algorithms chosen for implementation and testing in Twill are reviewed in depth. Supporting frameworks such as the Program Dependence Graph (PDG), Solution Set, and Solution Node will be illustrated. Following the frameworks, the Tabu Search Simulated Annealing (TSSA) algorithm's use of a combined Tabu Search (TS) and Simulated Annealing Neighborhood Generator (SANG) is expanded upon. After TSSA, the Genetic Search (GS) algorithms design with generations, genomes, mating, and mutation will be covered. Finally Twill's original Accumulator algorithm is illustrated. In Section 5, the way these algorithms are scored and judged is described. Afterwards in Section 6 the implementation of TSSA and GS will be explained.

4.1 Partitions

4.1.1 Solution Set

A solution set is a collection of hardware-software implementations of the same function that LLVM is operating on. A solution set is made of solutions that may be connected via a parent/child relationship, or through a different kind of hierarchy. The connections make it possible to move across the solution set, treating it as a graph. This let's search algorithms such as Tabu Search move across the space of generated hardware-software solutions. Each solution is encapsulated within a solution node.

4.1.2 Solution Nodes

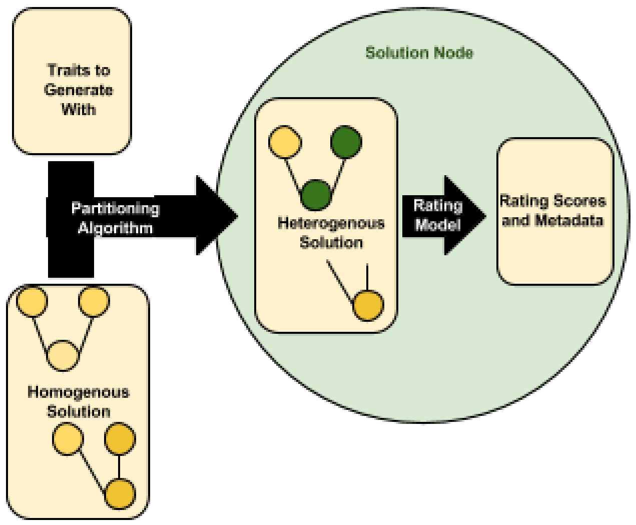


Figure 13 Solution Node Construction and Metadata

Solution nodes contain the actual hardware/software solution along with a collection of metadata, seen in Figure 13. The metadata is generated by cost model calculations, and by partitioning algorithms. Included inside metadata are the fitness and performance scores, along with hardware and software counts, total instruction counts, and time taken to generate the solution. Flags and other notes for search patterns to use while traversing the solution set are included as well. One important flag for example is the "tabu" flag, as it will help dictate the tabu search.

4.2 Tabu Search Simulated Annealing Algorithm

4.2.1 Overview

The Tabu Search - Simulated Annealing algorithm is designed to find the best result it can find with a local search space and allowed time. The algorithm is split into two parts, named the Simulated Annealing Neighborhood Generator (SANG) and the Tabu Search (TS): together they are the TSSA algorithm. This design was intended to leverage the power of an abstract tabu search onto a solution aware simulated annealing pass. While

SANG examines the actual instructions, the TSSA Tabu pass only compares rated performance which can be represented with a single number via the TSSA cost model, making the Tabu Search problem agnostic.

This algorithm was chosen because it has two different algorithms to use, SANG and TSSA (Tabu Search over SANG), and due to it’s simplistic qualities. The pseudo-random evaluation was easily realizable, and had no ordering or sorting of a solution. This means that processing power will not be wasted attempting to order some set of data, and instead it can be used to generate more solutions.

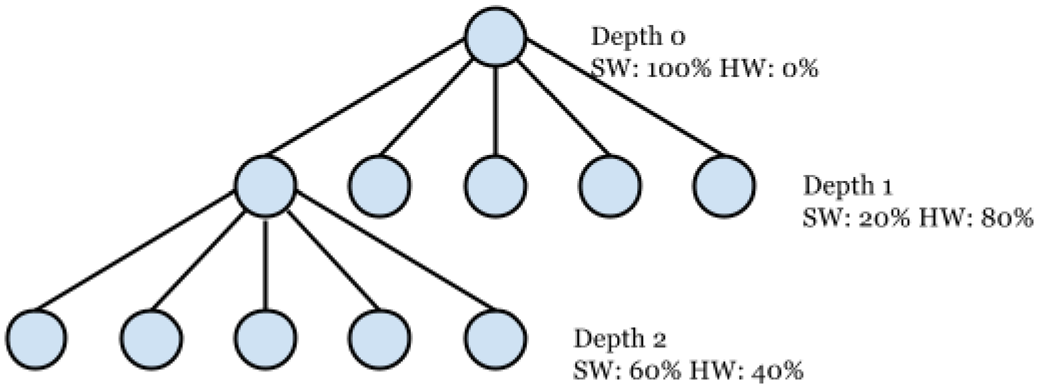


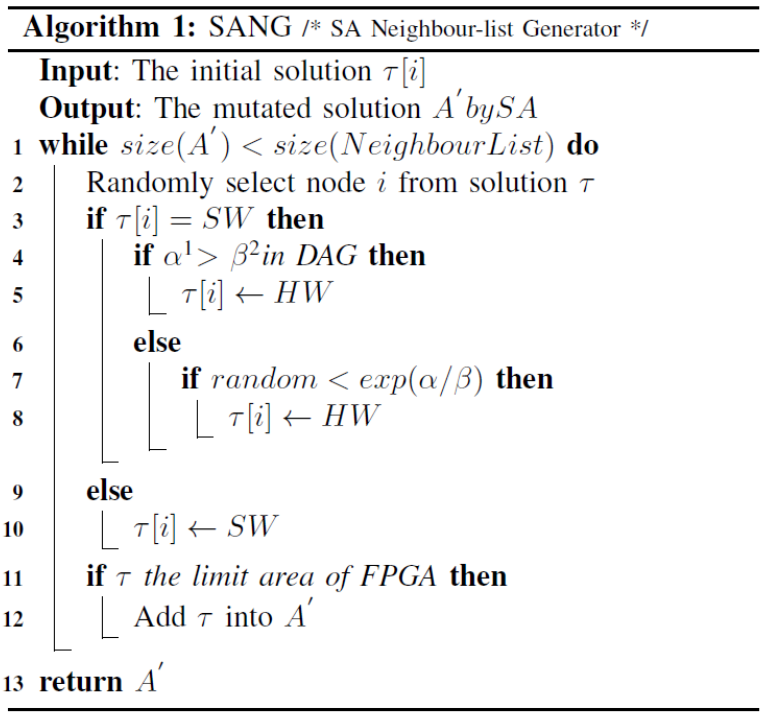
Figure 14 Parent Child Inheritance Inside a SANG Graph

The parent-child inheritance (shown in Figure 14) of the algorithm is promising as well, as it attempts to guarantee that the stronger solutions will be further improved with every other iteration of the algorithm.

It is assumed that this algorithm will perform well, and that it’s pseudo-random greedy search will find a greedy, but acceptable solution. It's rapid generation of varied solutions with a localized search appears to be well suited to the problem of hardware software heterogeneous solutions.

4.2.2 Simulated Annealing Neighborhood Generator

To explore related solutions, neighbors of a given solution can be created using simulated annealing. SANG generates a given set of annealed solutions from one parent solution. Two trait variables are passed on with permutations from the parent, the starting temperature and the cool down speed. The solution starts out at a given temperature, and with the initial settings. Each instruction is then examined, with it's costs being estimated. If the path cost is above a given threshold, it will automatically set the examined node into hardware in an attempt to alleviate the software path cost. If the past cost is still below a given threshold, but if a random [0,1] outcome is greater than exponential function using the change in cost and current temperature, then the move to hardware will still be made. Any SCC instructions that were initially hardware are reset to software as well, effectively inverting the solution from the parent.



```
Algorithm 1: SANG /* SA Neighbour-list Generator */
Input: The initial solution $\tau[i]$
Output: The mutated solution $A' bySA$
while $size(A') < size(NeighbourList)$ do
    Randomly select node $i$ from solution $\tau$
    if $\tau[i] = SW$ then
        if $\alpha^1 > \beta^2 in\ DAG$ then
            $\tau[i] \leftarrow HW$
        else
            if $random < exp(\alpha/\beta)$ then
                $\tau[i] \leftarrow HW$
    else
        $\tau[i] \leftarrow SW$
    if $\tau$ $the\ limit\ area\ of\ FPGA$ then
        Add $\tau$ into $A'$
return $A'$
```

Figure 15 SANG Algorithm [36]

This greedy algorithm is self-enclosed and can be run multiple times with various tweaks to it's traits, criteria, and thresholds. The Alpha and Beta values seen in Figure 15 are supposed to be cost oriented values, and can be set to any desired criteria. Once neighbors have been generated it's possible to explore a graph of possible HW/SW partitions using the tabu search, and refining of this search can be done with further passes of SANG on any given tabu solution. Evaluating the generated solution is done by calculating the communication costs.

$$c(P(v), v) = \begin{cases} \sum_{u \in P(v)} c(u, v), & \text{if } u \text{ is a SW task,} \\ \max_{u \in P(v)} \{c(u, v)\}, & \text{if } u \text{ is a HW task} \end{cases}$$

Figure 16 SANG Communication Costs [36]

$$Perf = Max\{T_{soft}, T_{hard}\} + Pena$$

Figure 17 SANG Solution Rating (Performance) [36]

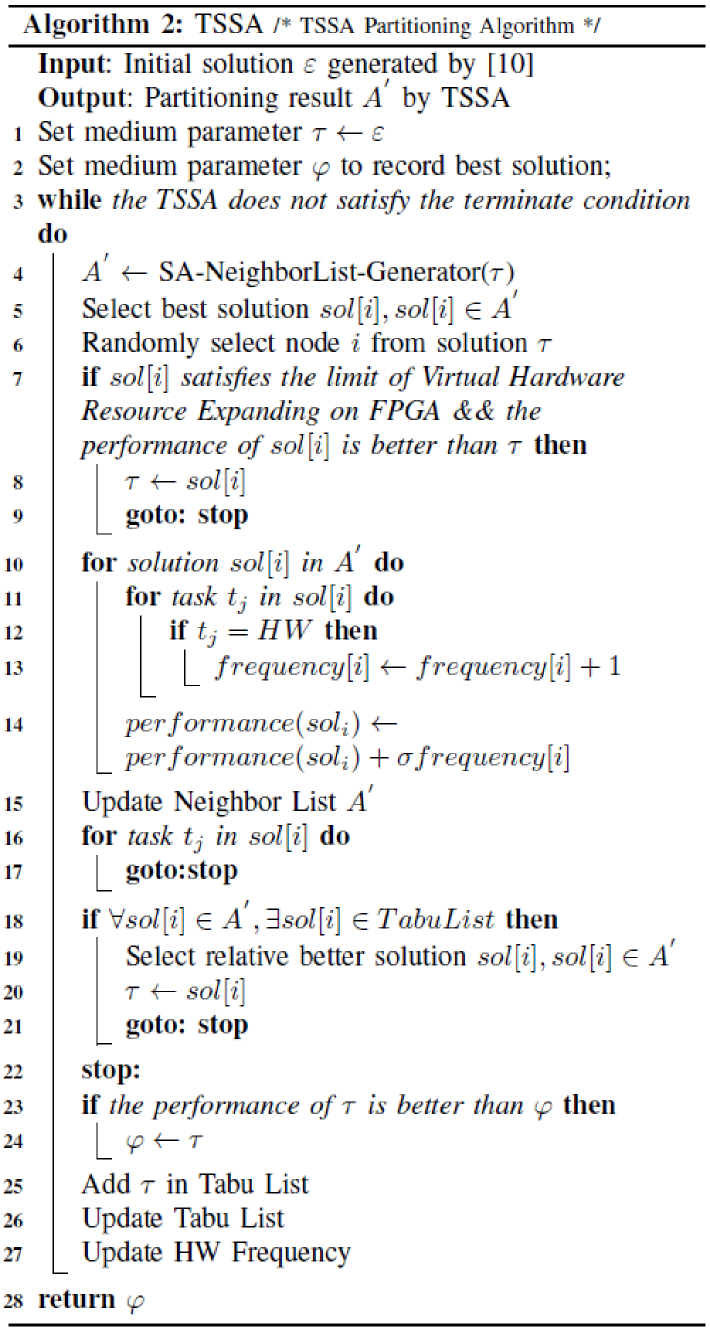
As noted, simulated annealing has a number of different control points. The initial temperature (the entropy of the system), and the cool down factor (how fast the system settles down), are easily controllable. They also generate very well defined results, giving us consistent output along a probability curve. The size of the neighborhood generated can be modified as well, making SANG feasible on machines of any power, and in turn letting machines that have extra resources easily create a larger neighborhood. SANG also inverts the previous entry (only SW instructions are allowed to become HW instructions, and any HW instruction become SW instructions again). This inversion lets two opposite solutions be evaluated rapidly, so that an optimal approach can be reached from both solution sides (all HW vs all SW).

4.2.3 Pre-Generated SANG Maps

For testing, SANG solutions will be pre generated in a massive tree so that the TSSA algorithm can be "tracked" as it moves through the previously generated space. With only a few iterations, this process is sufficient to flood memory, disk space, and processing time of many systems, but it will give us an omniscient view of an algorithms performance.

4.2.4 TS

The second half of the TSSA algorithm is the Tabu Search. It can be seen in Figure 18. TS starts with an initial solution node, which is automatically a tau solution node. Tau solution nodes are the "winning" solution node. The initial solution node is passed into SANG to generate child solution nodes with similar traits (plus permutations to give variety). The best solution out of all of these solution nodes is found, becoming the tau node. At this point, the tau solution node is compared against the gamma solution node, which is the "output" of the Tabu Search. If there is no gamma node, the tau node becomes gamma by default. If there is a gamma node, the two are compared and the winner is declared the gamma node. Regardless of this outcome the tau node is declared tabu from here on out. This means that it is put into a list that every potential tau node is checked against later in the tabu search process. This tabu list can eventually decay, so that older entries may become not tabu after some given number of cycles. After the tabu list has been added to and upkeep has been orchestrated, a new neighborhood is generated using SANG to permute the traits of the first node generated by the previous SANG pass. The whole process then starts over again, using this first node as the new "initial node", until it is deemed by some user defined end condition.



```
Algorithm 2: TSSA /* TSSA Partitioning Algorithm */
Input: Initial solution $\varepsilon$ generated by [10]
Output: Partitioning result $A'$ by TSSA
Set medium parameter $\tau \leftarrow \varepsilon$
Set medium parameter $\varphi$ to record best solution;
while the TSSA does not satisfy the terminate condition do
    $A' \leftarrow$ SA-NeighborList-Generator($\tau$)
    Select best solution $sol[i], sol[i] \in A'$
    Randomly select node $i$ from solution $\tau$
    if $sol[i]$ satisfies the limit of Virtual Hardware Resource Expanding on FPGA && the performance of $sol[i]$ is better than $\tau$ then
        $\tau \leftarrow sol[i]$
        goto: stop
    for solution $sol[i]$ in $A'$ do
        for task $t_j$ in $sol[i]$ do
            if $t_j = HW$ then
                $frequency[i] \leftarrow frequency[i] + 1$
        $performance(sol_i) \leftarrow performance(sol_i) + \sigma frequency[i]$
    Update Neighbor List $A'$
    for task $t_j$ in $sol[i]$ do
        goto:stop
    if $\forall sol[i] \in A', \exists sol[i] \in TabuList$ then
        Select relative better solution $sol[i], sol[i] \in A'$
        $\tau \leftarrow sol[i]$
        goto: stop
    stop:
    if the performance of $\tau$ is better than $\varphi$ then
        $\varphi \leftarrow \tau$
    Add $\tau$ in Tabu List
    Update Tabu List
    Update HW Frequency
return $\varphi$
```

Figure 18 Tabu Search Pass for the TSSA Algorithm [36]

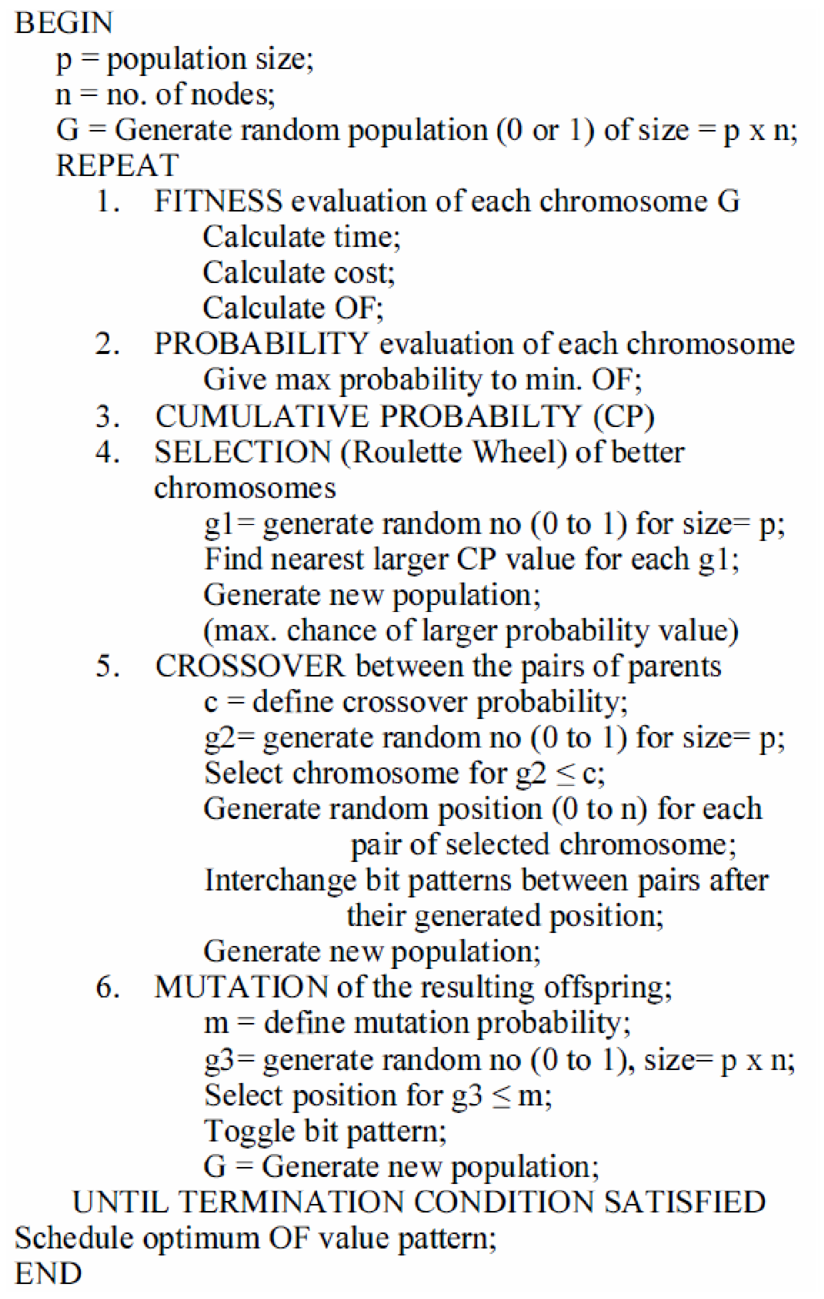
The idea of a tabu node is the crux of the Tabu Search: a tabu solution node means that it will never again look at nodes having the same traits as the tabu (since it has already evaluated nodes of that type) node. This makes TS a greedy style search focused on reducing the HW/SW graph as fast as possible. TS also engages in "following" lucrative solutions, as the best solution out of a neighborhood will become tau, possibly gamma, and will be used as the parent for the next generation of solutions.

Tabu search is localized, meaning that it can iterate repeatedly without consequence to memory, thus the only factor it has to worry about is time. Like SANG, TS can also be limited on how much effort it puts into refining it's search.

4.3 Genetic Search Algorithm

4.3.1 Overview

The Genetic Search algorithm is designed to find the best result with a local search space and allowed time. The algorithm is split into two parts, a realization of a generation's worth of genomes into solutions and the creation of a new generation based on the best solutions from the previous generation. Unlike TSSA there is no two part generation + search algorithm, only multiple generations intended to both increase diversity and hone in on desired traits.



```
BEGIN
   p = population size;
   n = no. of nodes;
   G = Generate random population (0 or 1) of size = p x n;
   REPEAT
      1.   FITNESS evaluation of each chromosome G
              Calculate time;
              Calculate cost;
              Calculate OF;
      2.   PROBABILITY evaluation of each chromosome
              Give max probability to min. OF;
      3.   CUMULATIVE PROBABILTY (CP)
      4.   SELECTION (Roulette Wheel) of better
           chromosomes
              g1= generate random no (0 to 1) for size= p;
              Find nearest larger CP value for each g1;
              Generate new population;
              (max. chance of larger probability value)
      5.   CROSSOVER between the pairs of parents
              c = define crossover probability;
              g2= generate random no (0 to 1) for size= p;
              Select chromosome for g2 ≤ c;
              Generate random position (0 to n) for each
                          pair of selected chromosome;
              Interchange bit patterns between pairs after
                          their generated position;
              Generate new population;
      6.   MUTATION of the resulting offspring;
              m = define mutation probability;
              g3= generate random no (0 to 1), size= p x n;
              Select position for g3 ≤ m;
              Toggle bit pattern;
              G = Generate new population;
    UNTIL TERMINATION CONDITION SATISFIED
Schedule optimum OF value pattern;
END
```

Figure 19 Genetic Algorithm [22]

This algorithm in Figure 19 exhibits some randomness with mutations, but also displays the trait of common genetics with the idea of alleles, crossovers, and mutations. These traits have been proven to be beneficial with reaching an ideal solution in other problem spaces. Along with being a good solution generator, a genetic search also usually keeps good traits intact during crossover. This means that when a good trait is found, minimal changes to it may occur during crossover or mutations, but enough small changes may eventually yield a stronger solution.

The lack of reliance on randomness, and emphasis on persistence of good traits, makes this a great choice for a partitioning algorithm. This may generate a noisy solution, due to mutations, but by emphasizing the desired traits, a gradual progression towards an ideal outcome will emerge.

4.3.2 Population

The population will be controllable in size, as with the generations. However the "mating" process will be done by selecting the most fit solutions in a population and mating two into two children (from the results of a crossover) until the new population is the same size as the old population. This will ensure a "localized" style search in regards with memory constraints in the compiler, as a growing population could quickly cause a major lock up or memory failure. An exhaustive searches explored size is easy to calculate since the following equation will be true:

EXPLORED SOLUTIONS = LIMIT_GENERATION * LIMIT_POPULATION

4.3.3 Genomes

The genome was constructed to be a bit pattern that would correspond with the assignment of encountered instructions. Stored as an array, it was easy to splice, mutate, and move along the bit pattern. The bit pattern must be sufficiently sized to have unique results for each encountered instruction. This requires knowing how large the input code sample, which only needs to be calculated once during use of the genetic algorithm.

4.3.4 Crossover

Crossover of genomes mimics biological crossover, where chromosomes separate at given indices. Then “genetic information” (which is the HW/SW partition assignments) can be swapped between two chromosomes with the same chromatids, but with different alleles (which lead to some different trait outcomes). Two genomes of the same size separate at the same points along their length, called the loci. If one genome is called A and one B, one part of A is merged with its complementary B part, and the versa occurs where the left over B part merges with the complementary A part. The crossover point is defined to be random.

4.3.5 Mutation

Driving genetic diversity, mutation occurs along with crossover to ensure that “fresh” combinations of alleles are created, so that a population will not stagnate, thus allowing evolution, as opposed to blind refinement. Blind refinement would lead to whole populations becoming trapped in a local minima/maxima.

Earlier in generations, it can be popular for mutations to be a bit more aggressive, as a good solution has not shown itself yet, so many different possibilities must be explored quickly. As the best solutions begin to emerge, the mutations begin to slow down so that the crossover function can refine the solution.

Mutations are done by randomly indexing a genome and flipping the bits found after that index. This can be done any number of times.

4.3.6 Fitness

For each solution there is a total fitness that can be calculated by using the genome. Total fitness is used to evaluate one member of a population against another, and it is calculated through a collection of costs. Fitness calculations will be covered in Section 5.

4.3.7 Parameters

The following areas of the genetic algorithm were modulated in order to change the time required to obtain a solution.

- LIMIT_GENERATION - The number of "iterations" the population will go through. By defining this as a modulated value, multiple iterations of the genetic algorithm can be run with varying generation limits to determine the effectiveness of 20 generations as compared with 5.
- LIMIT_POPULATION - The amount of solutions each population will have. Like LIMIT_GENERATION this value is also modifiable.
- LIMIT_ALLELE - The length of the genome for each solution
- LIMIT_MUTATIONS - The amount of mutations the initial generation starts with
- DEC_MUTATIONS - The amount mutations decrement by each generation. Unlike the limits, this value will change how fast the mutations will persevere for (LIMIT_MUTATIONS/DEC_MUTATIONS) = max generations.

The genetic algorithm follows the following loop complexity:

```
FOR EACH GENERATION
FOR EACH POPULATION
    FOR EACH ALLELE
N*LOG(N) POPULATION SORT
    FOR EACH POPULATION
        FOR EACH POPULATION * 2
        FOR EACH ALLELE * 4
FOR EACH MUTATION
```

Which gives us the following rough equation for operations needed:

*OPS = (GEN * POP) (LOG(POP) + ALLELE * (1 + 16 POP * MUTATION)*

Using a starting base of 20 generations with a population of 20 and an allele size of 1 with 1 mutation each cycle we get the following growth in our system.

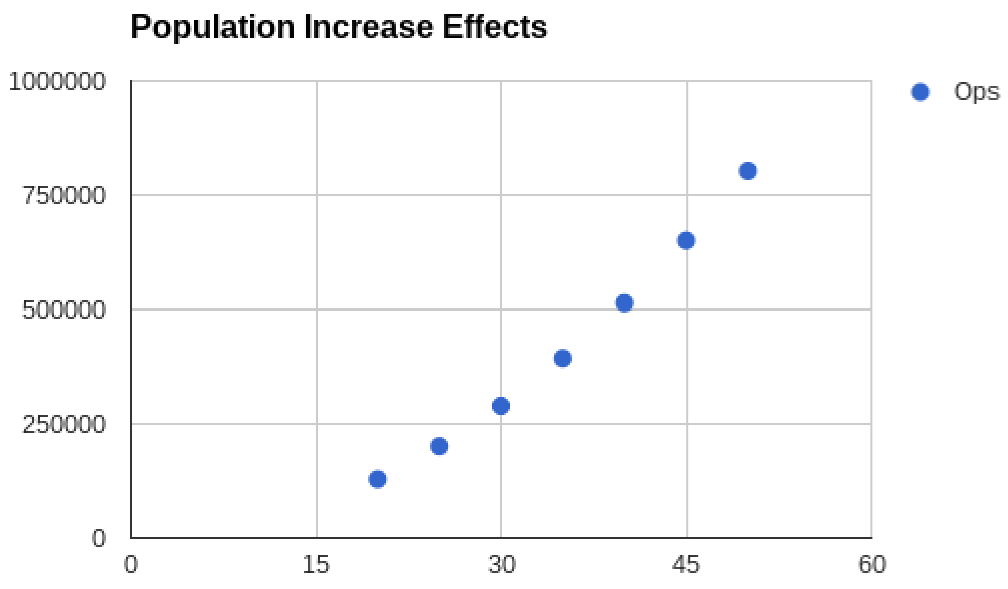


Figure 20 O(n^2) Population Increase

As expected in Figure 20, the generational increase results in a linear growth, but the population increases show that we have a complexity of O(n^2).

4.4 Twill's Original Accumulator

Twill's original accumulator design focuses on the Knapsack idea of packing a limited space. The space in question is a percentage of the total code's software latency time. Each instruction contributes a certain amount to this latency, as illustrated in Figure 21.

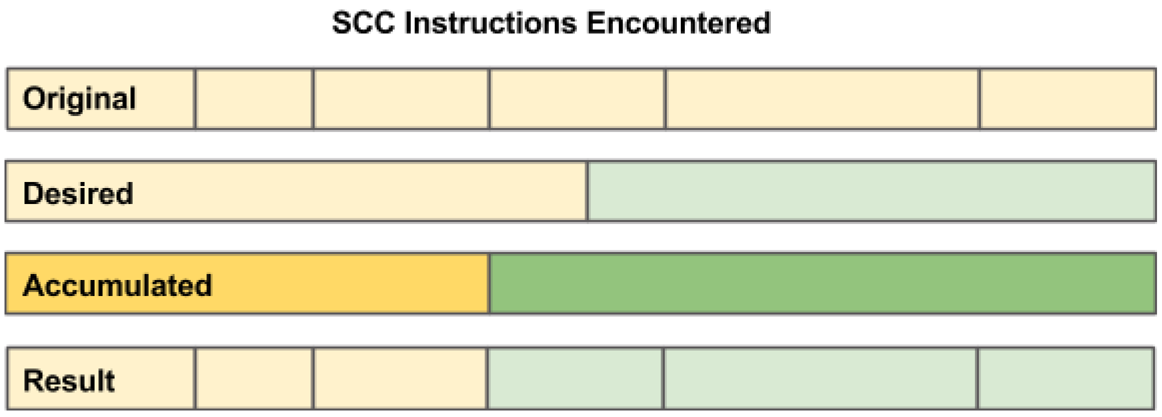


Figure 21 Illustration of Accumulator HW/SW Partitioning

Out of the original instructions, with their delay times, the whole program can be visualized as a bar. The desired percentage to be in software (yellow) as opposed to hardware (green) is set in the desired bar. Since the division does not fully take the fourth instructions latency up, it does not become software, but rather becomes hardware. This threshold behavior is induced because an instruction cannot become partially hardware or software, so a decision must be made one way or the other. The resultant instructions are then designated according to the Accumulated outcome.

5 Cost Models and Criteria

Each algorithm comes with its own cost model, for the Tabu Search Simulated Annealing (TSSA) it is the Performance Rating and for the Genetic Search (GS) it is the Fitness Rating. These rating systems are described with regards to their design and expected implementation. Along with the rating systems, other methods of evaluating these algorithms are explained, including time to find a solution, RAM requirements during runtime, and cuts across the Program Dependence Graph (PDG) created for partition. In Section Six, the Heuristics (Section 4) and Ratings (Section 5), are implemented in Twill. Section Seven will cover the testing and results of each algorithms implementation in Twill.

5.1 Evaluation

To evaluate the algorithms, a rating and cost model is needed. The cost model will judge the partitions using costs will estimate costs such as latency, area, and power costs. The costs must be defined for the compiler, with information about hardware and software costs for various instructions. Then the cost model will be run across the partition after the partition has been generated. The separation of the cost model and partitioning algorithm means that different cost models can be explored using one single method of partitioning.

For this work, we decided that every algorithm's cost model would be used on each algorithm. For example, a Genetic solution would be rated with a Genetic Fitness and a TSSA Performance rating together. This way an algorithm that produces a good solution across the board can be seen as a "strong" solution, while an algorithm that produces a solution that has contradicting ratings may be a result of a bad cost model.

5.2 **Costs**

For all cost models, costs are used. These help the model estimate power, area, and timing usage. There are four major costs that are used.

- SW_TIME - The software time for a given instruction
- HW_TIME - The hardware time for a given instruction
- SW_COST - The software cost for an instruction
- HW_COST - The hardware cost for an instruction

While cost is a very general description, papers insisted on using it. Cost in this term can mean power usage, area usage, or other cost factors. Many algorithms discern time as its own unique cost since heterogeneous solutions focus on reducing processing time along with one another dimension, hence the nebulous names of HW_COST and SW_COST.

SW_TIME is found by using the calculated latency generated by an instruction. HW_TIME will use the same latency, but will divide by a constant to represent the speedup created by putting an instruction in dedicated hardware (no pipeline/etc). SW_COST is a more nebulous cost, and it was decided that this would represent the area usage of SW. The area usage of SW represents the static processor, while the HW_COST is the LUTs needed to implement the custom logic.

5.2.1 Area Costs

With SW_COST and HW_COST values required, it was decided that the costs would be relative. SW_COST was designated to imply the area cost of a whole processor’s worth of silicon or the divided cost of a whole processors worth with regards to the amount of instructions in software. The second SW_COST designation means that if five instructions are implemented in software, the total cost of the processor area wise will be split across those five instructions. This style is done to entice models to penalize extremely low SW assignments, where in the worst case no instructions are implemented in software so a whole processor is sitting there as dead silicon. Now with this area of processor linkage to SW_COST, the HW_COST naturally will the area needed to implement the given instruction in hardware. This can be done by examining the size of the data used in the instruction and the operation. If the instruction is basic, such as an add or subtract, the HW_COST will be low, but if it is a division or multiplication, it will be higher. Along with the operational HW_COST, the width will play a role in the HW_COST as well. With these definitions of HW_COST and SW_COST per instruction, and the time related HW_TIME and SW_TIME definitions, an algorithm can correctly estimate the performance of a solution, and in turn correctly follow the contours created by a search across a set of given solutions.

5.3 Cost Models

5.3.1 Performance

Brought forth in the TSSA algorithm, a performance rating can be calculated for a given solution. The core equation is as follows in Figure 22.

$$\text{Performance} = \text{MAX}(\text{Tsoft},\text{Thard}) + \text{Penalties}$$

Figure 22 Performance Rating Calculations

Higher times and higher penalties result in a higher score. Heterogeneous cosynthesis is all about minima, so it makes sense that our evaluation metric follows the same minima focused calculation. This may make it difficult to represent as a low "performance" is seen as desired outcome.

The first maximum evaluation between SW time and HW time is the comparison of the total time for the SW and HW partitions. Penalties are more involved, since they focus on previous instructions to evaluate the communication costs. Communication costs are calculated in the following manner inside Figure 23.

$$\text{Penalties} = \begin{cases} \text{SUM}\ (u, v) & \textit{if}\ \text{u is SW} \\ \text{MAX}\ (u, v) & \textit{if}\ \text{u is HW} \end{cases}$$
$$v = \textit{previous penalties}$$

Figure 23 Performance Rating Communication Costs

5.3.2 Fitness

Fitness ratings originate from the Genetic Algorithm, and examine the HW and SW costs/timing results. Unlike the Performance algorithm, communication costs are not evaluated, and SW costs are considered only once. This is due to the fact that software costs as the area needed (a processors' area does not change) along with hardware's area costs. These calculations are seen in Figure 24 and 25.

```
BEGIN
Take one chromosome
Cost = 0; dummy = 0;
for i=1:n
   if (pattern[i] == 1)
            execution_time[i] = hw_time[i];
           Cost = Cost + hw_cost[i];
   else
            execution_time[i] = sw_time[i];
            if (dummy == 0)
                 {Cost = Cost + sw_cost[i];
                 dummy = 1;}
Start_time[1] = 0;
End_time[1] = execution_time[1];
for i=2:n
   Start1 = max{End_time[predecessor_task]};
   for j=1:i-1
            if(pattern[i] == pattern[j])
               z=j;
   end
   Start2 = End_time[z];
   Start_time[i] = max(Start1 + Start2)
   End_time[i] = Start_time[i] + execution_time[i];
end
Time = End_time[n];
Cost;
END
```

Figure 24 Fitness Rating Calculation for a Given Solution

The cost variable constantly accumulates hardware costs, but will only have the software cost added once. This represents the processor (since SW instructions found later on will run on the same processor) opposed to the custom hardware needed. The time for both hardware and software is loaded into the execution time array, and is then injected into start time and end time arrays. The patterns represented here are the genomes used in our algorithm.

After the Cost and Time have been calculated, we then calculated the fitness.

$$\text{fitness} = \begin{cases} k1 * (\text{time} - \text{tr}) + k2 * \text{cost} & \textit{if } \text{time} > \text{tr} \\ \text{cost} & \textit{if } \text{time} < \text{tr} \end{cases}$$

Figure 25 Fitness Rating Calculation

Time Restriction (tr) is a constraint deadline that we use to shape the fitness calculation. The K1 and K2 values will be used to bias the fitness calculations to make cost or timing become more expensive. Based on the accumulation of time calculated from the cost model in Figure 26, whether it is below or above the tr constraint in Figure 27 will dictate what factors go into the fitness rating.

5.4 Test Code

In order to test the algorithms, input code must be used. The CHStone benchmark [37] for High Level Synthesis was selected to become part of the test code because of it's intention to be an HLS benchmark. Along with CHStone tests a variety of smaller samples were selected to have smaller pieces of code that would be used during debug and polishing of the implemented algorithms.

5.4.1 CHStone Media Processing

With video being a prominent use of blended solutions, media processing was a major focus in the CHStone Program Suite. For the set of media tests a "Linear predictive coding analysis of global system for mobile communications" (GSM) was included along with JPEG image decompression and MPEG-2 motion vector decoding. The JPEG code sample had the most lines of C code at 1,692 lines and 1,029 addition/subtraction operations. It also contained the most branches, with 213 if statements, 64 switch statements, 90 for loops, 27 while loops, and 228 breaks.

5.4.2 CHStone - Security

Along with media processing, security applications are also of interest in the field of heterogeneous computing. The AES, Blowfish, and SHA encryption standards were included as well. SHA and Blowfish both had a light amount but a large amount of logic and shifting operations. AES came in with the largest amount of shifts, with 758 operations across its 716 lines of code.

5.4.3 CHStone - Intentions

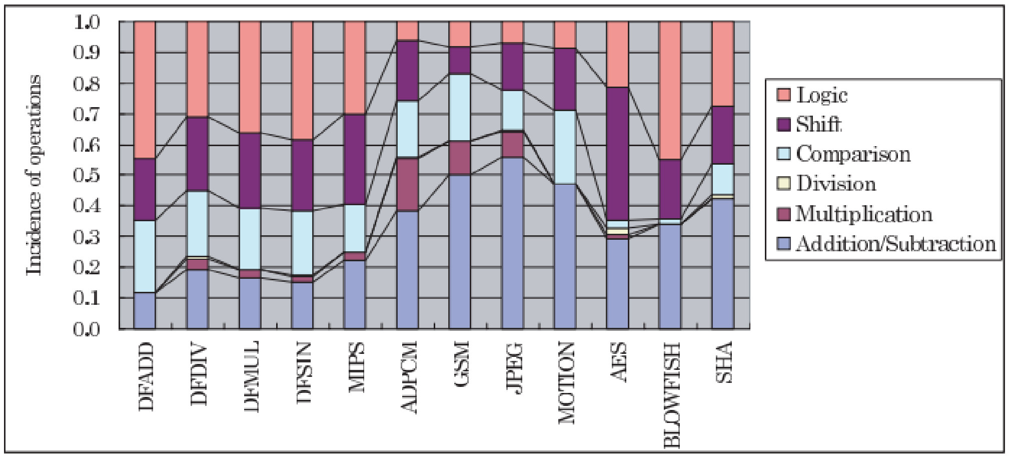


Figure 26 CHStone Operation Distributions [37]

CHStone intends to cover a variety of program styles with regards to the types of operations a program may include, as shown in Figure 26. As noted in the previous sections, AES has a high amount of shifting compared to the other algorithms, and JPEG/GSM have sizeable amounts of multiplication. Division is fairly uncommon, except for the AES implementation. Along with a visualization of the operations, it is also interesting to see the distribution of control flow statements.

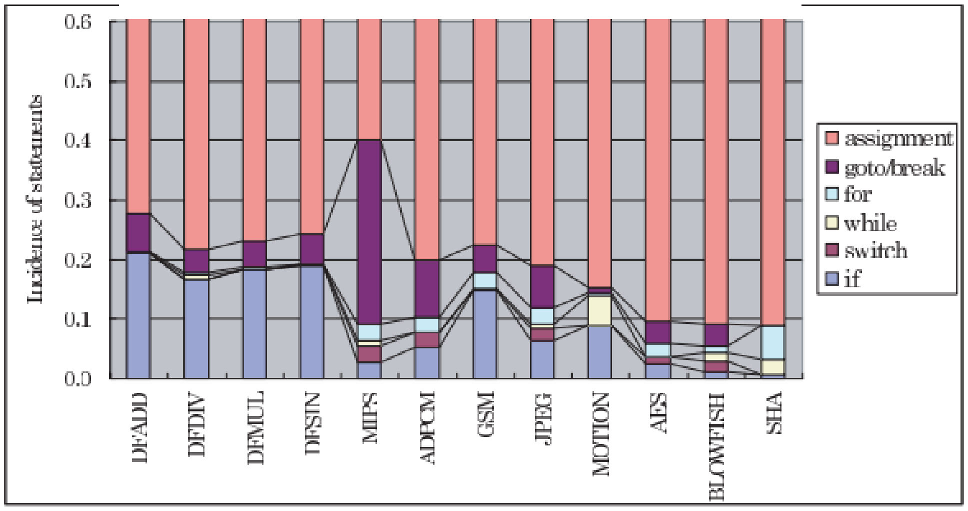


Figure 27 CHStone Branch Distribution [37]

Shown in Figure 27, the control flow statements of the CHStone programs are dominated by assignments, to the point where the Y axis of this graph only reaches 60% of the program, since the other 40% is assignments across the board. MIPS is extremely heavy on control flow changes, with many goto/breaks. Apart from MIPS, the rest of the algorithms have a fairly low distribution of goto/breaks, but the encryption and media processing algorithms contain sizeable amounts of loops.

5.5 Criteria

To evaluate the performance of the algorithms, some data points must be used to empirically compare them. The following points of interest were devised to evaluate the algorithms' performance.

- Cost Models
  - Total Performance Rating
  - Genetic Fitness Rating
  - Correlation of Performance / Fitness

- Resources Needed to Find Solution
    - CPU Time Needed
    - RAM Needed
- Resultant Solution Traits
    - Number of Cuts in the Program Dependence Graph (PDG)
    - Implemented HW/SW Percentages
    - Resultant Queues from Cuts

These criteria focus on a number of factors, but first the ratings generated by the cost models are extremely important. These ratings will dictate the solution generated by the compiler, if the rating system produces inconsistent results then the ratings cannot be trusted. Once the rating system has been verified to be consistent then the solutions can be checked against the ratings with factors such as the cuts generated in the PDG. The cuts calculated can also be examine with the queues actually generated in the ending .v/.c resultant files generated by Twill.

A correlation of Performance and Fitness is also important, as this will show whether or not evaluations of solutions from one algorithm will still be "good" in the eyes of another algorithm. If Fitness is positively correlated with Performance, this means that a good fitness rating means the solution will have a good performance rating.

Along with the solution reached, the time required to obtain the solution is also important. Many papers go into detail about this aspect of heterogeneous computing, since this problem space is considered to be NP-hard. Resultant ratings will be compared with the time necessary to determine if the time to compute the solution is "worth" the time. This "worthiness" also goes for memory usage, but most of these solutions do not have memory issues as they focus on a local search, except for an exhaustive SANG/random partition generation.

The amount of cuts in the graph is also important as the more cuts there are, the more hardware is generated to handling enqueueing/dequeuing. With each hardware-software crossing, software is also generated on the other side as well. A solution that generates a minimal amount means that communication costs will be minimal, and a solution that has a high amount of queues means that the algorithm that generated it may not be as elegant as intended once the communication costs come into play.

6 Implementation

Both the Tabu Search Simulated Annealing (TSSA) and Genetic Search (GS) are implemented in Twill, with notes and diagrams showing how it was integrated inside the LLVM Twill Transforms. Along with the heuristics implemented, the ratings, cost models, and representations of a partition solution node and solution graph are also illustrated. Following this implementation, Section Seven displays the results of both TSSA and GS, with Chapter Eight summarizing and concluding the thesis.

6.1 Partition Representation

6.1.1 GraphNodes and GraphNodeLists

To create "nodes" for the graph of possible partition outcomes, a new set of classes was created: the GraphNode class. The GraphNode is the implemented version of a Partition Solution that is intended to be used inside an LLVM transform. Inside GraphNode exists the following:

- A HW and SW partition - This is required, and is the main focus of the partitioning algorithms
- Traits for SANG
- Genome for the Genetic Algorithm
- Performance Rating - TSSA-based rating of HW/SW partitions
- Fitness Rating- GA-based rating of HW/SW partitions
- HW instruction count - Instructions in Hardware
- SW instruction count - Instructions in Software

These are mainly represented with ints and doubles, as they are scalar values. The genome and partitions are represented in vector format.

6.2 LLVM Transform Additions

Leveraging the DSWP Transform that was implemented in LLVM as a part of Twill, additions were made to the location where Twill's Accumulator partitioning method was called. For the thesis it was planned that the Genetic Algorithm, original Twill algorithm, and the TSSA algorithm would all run across the same PDG so that their results with the same input code could be captured.

6.2.1 Genetic Search

The genetic search was enclosed by three for loops to capture solutions for ranges of population, generation, and mutation counts. Enclosed within, it was assumed that population, generation, and mutations were properly set. A for loop ranged from zero to the number of decided generations. A vector of population genomes exist with an equal vector of GraphNodes within. The genomes are included inside the GraphNodes, but it was decided to have a "children" copy so that the current GraphNodes would not be modified. Following the population variables defined, a function generates the GraphNodes according to their respective genomes. Within the function, the ratings are also calculated and added to the GraphNode. At this point there is now a free floating genome paired with a GraphNode that has the same genome within it along with a generated solution and ratings for the solution. The vector of GraphNodes is then ordered by best fitness. A biased random selector selects two of the best GraphNodes, and then performs a crossover of the two solutions' genomes. Since the genomes are arrays, this is done with some simple indexing. The new genomes are stored in the vector of genomes that is apart from the vector of GraphNodes. Finally the mutations are applied to each of the new genomes. The number of times a genome is mutated is defined earlier. Random points are chosen in the genome using a modulus operation and from that random point to the end of a genome the HW/SW bits are inverted. Following this mutation, two random solutions are selected again until the amount of new solutions is the size of the population. Out of all the current GraphNodes, the one with the best solution is compared against the "winning" node. If it is better than the winning node or if there is no winning node, it becomes the new winning node. The new genomes are then generated into full solutions and the process of rating, ordering, crossover, mutating, and comparison happens again for the number of generations that has been defined. After all the generations have been finished, the winning node is submitted as the best possible solution.

6.2.2 Accumulator

The accumulator was not changed, but the rating calculations were added to it's partitioning algorithm in order to rate the solutions. Along with adding the ratings, a for loop enclosed the Accumulator intended to range from 0 to 1 in selectable increments in order to capture the solutions for the whole range of possible Accumulator solutions.

6.2.3 Simulated Annealing

For the simulated annealing, three for loops were added to change the inital temperature, cooldown, and depth of the SANG graph. Since the SANG graph is built up using parent-child inhertiance it was important to create a graph generation system that would operate smoothly as the size of pending nodes to generate grew. A FIFO queue was created in order to store the parent nodes, and the first "head node" was set inside with the partition being 100% software (in order to represent the input of the code). A while loop checks the depth of the children it generates from a parent popped from the FIFO queue. If the depth is over the maximum depth, it will end and proceed to the Tabu Search. Otherwise, the children nodes are created using the popped parent node as a template. For simulated annealing, the parent HW/SW distribution is taken and inverted. A loop runs over all instructions. All HW instructions become SW instructions, and all SW instructions have a chance to become SW instruction as designated by a check. The check is whether a randomly generated number from 0 to 1 is less than the exponent of the delay cost delta of implementing the instruction in hardware over the current temperature. This is done with simple if statement branches. After each instruction has been evaluated, the ratings are calculated for the solution. Upon finishing the task the GraphNode then has children pointers (with a blank GraphNode class) created inside it and a parent pointer referencing its original parent. The depth of this new node is +1 the depth of the parent node. Following this creation of links for the whole SANG graph, the GraphNode is placed at the end of the queue. The next parent is popped off and processed as well, until the depth limit has been reached. For each node created, it can be compared against a "winning node". This node is distinct from the genetic winning node, and is called the SANGWinningNode in the code. The same comparison operation follows, and the best performance node takes the place of the SANG winning node. The original SW head node stays intact in its own variable so even after the queue is expended there is a graph with the original 100% software at the top that can be traversed by using the child pointers located in each GraphNode.

6.2.4 Tabu Search

Following the SANG graph generation, the Tabu Search will use the head node as a starting point in it's search. Any neighbors for a given node have been generated as well, but typically with the Tabu Search the neighbors would be generated upon examining a node. This keeps Tabu Search as a local search, as opposed to an exhaustive search like SANG. Starting at the head node, the head node is predeclared as a tau node, meaning it is the best node found in the current group of nodes examined, and since there is no gamma node, it becomes the gamma node as well (the best found across all of the Tabu Search). Per the rules of the Tabu search, it becomes off limits. The children around are added to an empty list of Nodes. This list will be used as a FIFO queue, and it will be filled up to a certain size (the number of children). The best possible node will be found out of the children using a basic performance comparison, and it's children will fill in the list. Any nodes that are present in the children list and the tabu list are eliminated. The best possible node out of the surviving is now designated as the new tau node, and it will be compared against the gamma node. If it is better than the gamma node, done with a simple if check, it becomes the gamma node. It is then added to the tabu list so nodes like it are avoided. This whole search pattern is inside a for loop, and will iterate as many times as is desired. Once the search has been run for the desired time, the gamma node, and the solution within, is presented as the best solution found.

6.2.5 Output

It was important to gather the data stored in the GraphNode for each solution found. The points of interest such as the HW count, SW count, performance rating, and fitness rating were printed out in a tab delimited format for processing later. Their traits were also printed as well, such as what generation they were from for genetic solutions, and what temperature the solution started at for TS/SANG solutions. Following the collection of this data the Genetic, Accumulator, SANG, and TSSA best solutions were pushed through the rest of the Twill compiler to see the queues created and output .c / .v files.

7 Results

With both the Tabu Search Simulated Annealing algorithm and the Genetic algorithm implemented in Twill, their characteristics, processing time, and results are examined. Along with these two new algorithms, the original Twill Accumulator partitioning algorithm's performance will be documented as well with the new rating systems used in both the TSSA and Genetic algorithm.

7.1 Combined Cost Models

Both cost models were implemented across all three algorithms, and modifications were made to the following variables. A selection of these runs have been shown below in Table 1. A variety of runs were used, but these tests show some of the search space. High hardware costs can be seen in test 3 and 4, with equal costs in test 5. The K1/K2 values also are weighted differently, with K1 changing the severity of the time portion of the fitness calculation, and K2 changing the cost portion of the calculation.

Table 1 Cost Model Configurations

TEST	HW_COST	SW_COST	HW_TIME	SW_TIME	K_ONE	K_TWO
1	2	3	2	6	0.3	0.1
2	2	3	2	6	0.3	0.3
3	5	3	10	6	0.1	0.3
4	5	3	10	6	0.3	0.3
5	10	10	10	10	0.3	0.3

Using these values, sample code was run and scores were taken from the three algorithms. In Figure 28 and 29 the fitness and performance scores are shown separately. It should be noted that the scores for the TSSA and Genetic algorithms are much smaller than the accumulator scores. Test 1 came out ahead for the Accumulator, and was designed to reward hardware placement (33% cheaper cost, 300% faster instructions) with an emphasis on timing (0.3 K1) rather than costs (0.1 K2) for the fitness model. Because of it's scores with the Accumulator, the settings from Test 1 were used for the rest of the thesis.

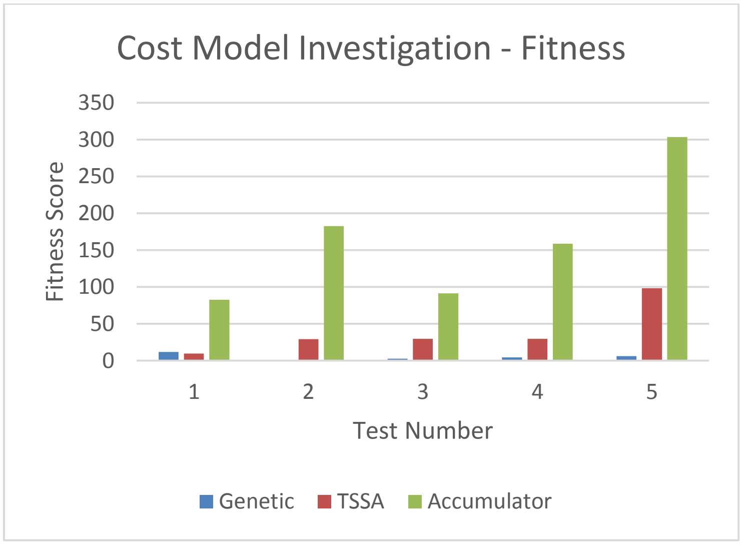


Figure 28 Fitness Scoring Outcomes from Cost Model Tests

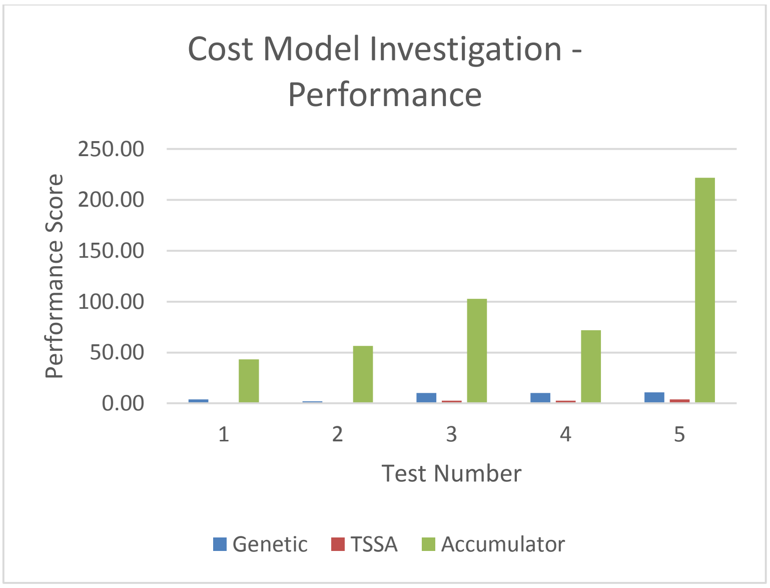


Figure 29 Performance Scoring Outcomes from Cost Model Tests

In deeper detail, the cost models can be examined looking only at the Genetic and TSSA outcomes. This is important since the the TSSA and Genetic algorithms are the two that we desire to compare the most.

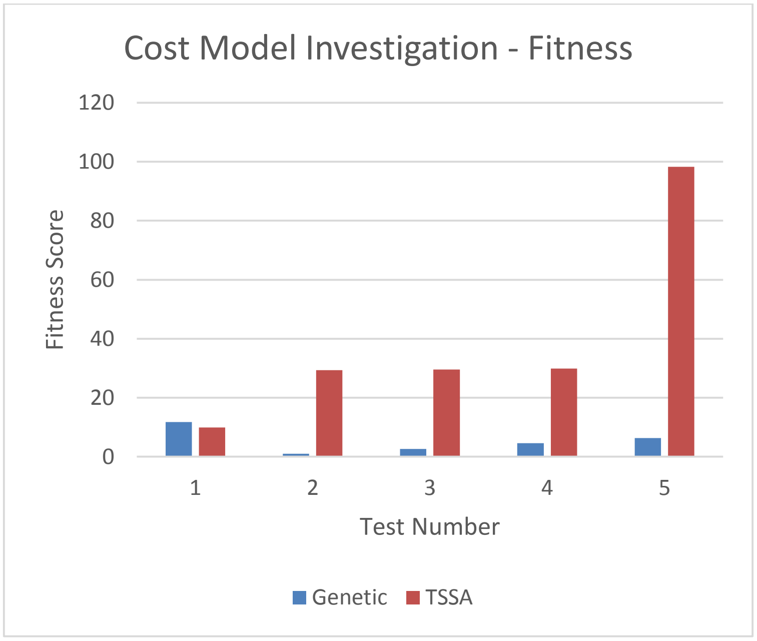


Figure 30 Fitness Scoring Outcomes minus Accumulator Results

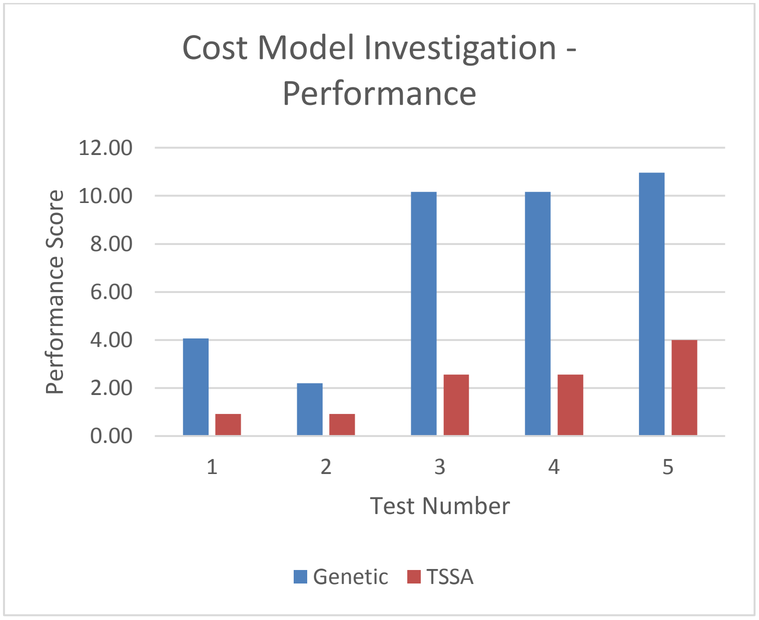


Figure 31 Performance Scoring Outcomes minus Accumulator Results

One interesting outcome in Figure 30-31 is how on the Genetic algorithm, TSSA still comes out ahead with Test 1, but with tests 2-5 it does not using the Fitness cost model. Note how the Genetic algorithm does not perform better than TSSA with regards to the Performance cost model. From here it was decided that TSSA would outperform Genetic regardless of changes to the parameters of the cost model with regards to the Performance cost model. Genetics' scoring in its own domain (Fitness rating) is unreliable since according to different cost parameters it either wins or loses. In all, this shows that the TSSA algorithm is able to consistently find a solution that the Performance cost model considers strong, showing that regardless of the changes to cost parameters, TSSA with Performance will still find a good solution according to the rules stated by the cost parameters.

7.2 Accumulator Results

With Twill's original accumulator, we can define a target percentage of SCC instructions to be in hardware. Ideally, Twill will generate a hardware partition exactly as large as the target percentage that has been defined. However testing of Twill's Accumulator Partitioning Algorithm found that the algorithm operates on latency thresholds. Latency thresholds are set by the user, and the cumulative latency of all the SCC instructions in hardware are summed together and judged against this threshold. This meant that while 30% hardware may be desired, there may only be a choice between 20% and 50%. This resulted in partition outcomes that would tend to follow a smooth sloping 0-100% instructions-in-hardware outcome, but would succumb to flat valleys where the threshold could not be crossed. The partitions generated could still be evaluated using the Fitness and Performance ratings however. The only issue to keep in mind is that a clean range could not be obtained by the design of Twill's Accumulator. Test code was run through the accumulator in order to see the shape that the cost models would generate under a single cut that moved across the original homogeneous partition. A very strong trend was observed in both the performance and fitness ratings with Figures 32-33.

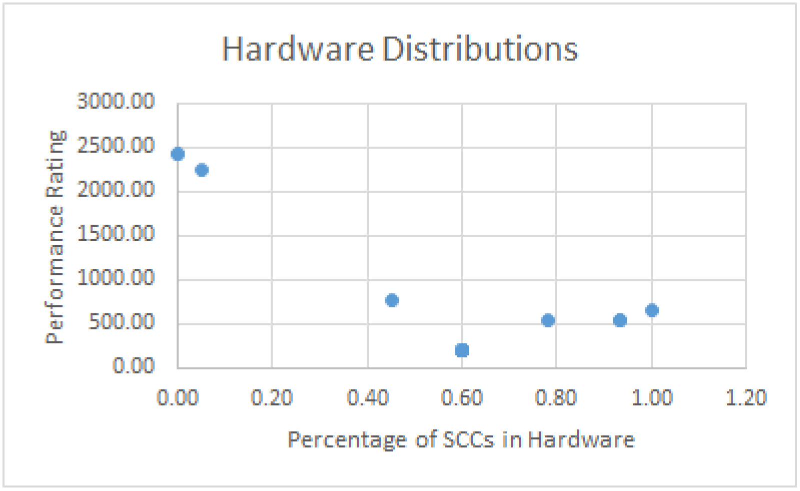


Figure 32 MIPS Performance Ratings Given Different Amounts of HW Implemented

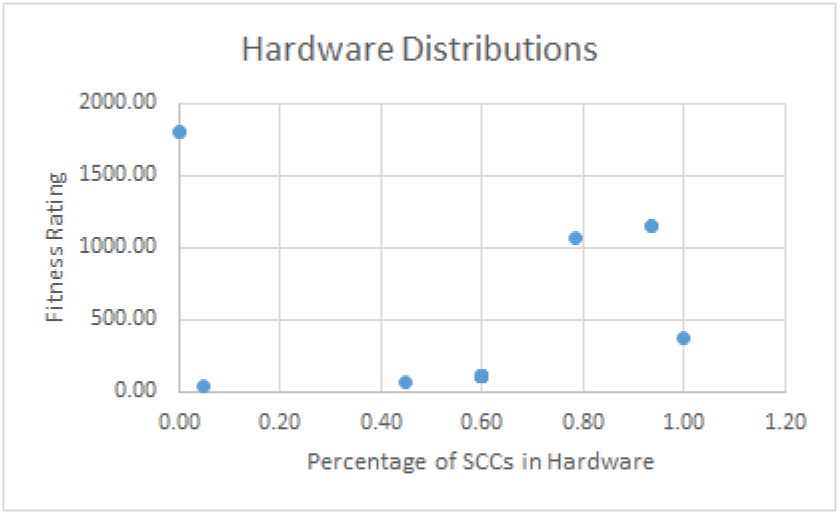


Figure 33 MIPS Fitness Ratings Given Different Amounts of HW Implemented

It is clear that both ratings have "sweet spots" where the best fitness and best performance can be achieved. As seen in the results of a MIPS simulator being the test case, the fitness minima can be seen where 60% of the SCCs are implemented in hardware. The performance maxima also has its sweet spot at this location, seen in Figure 34 and Figure 35.

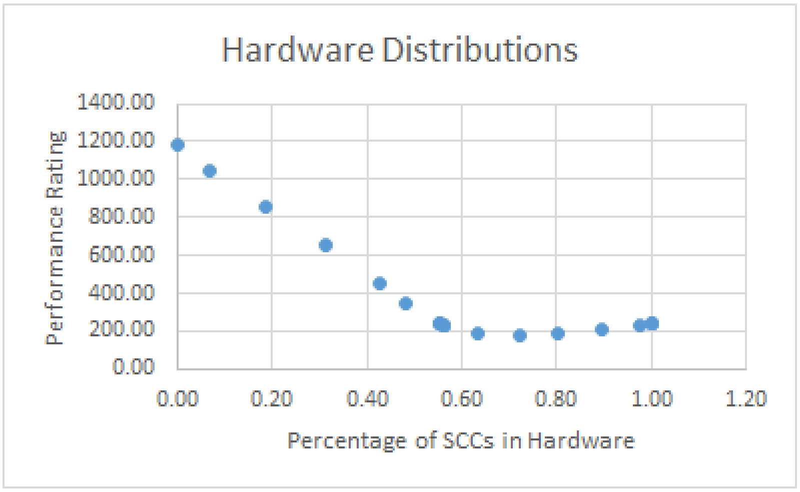


Figure 34 MPEG-2 Performance Ratings

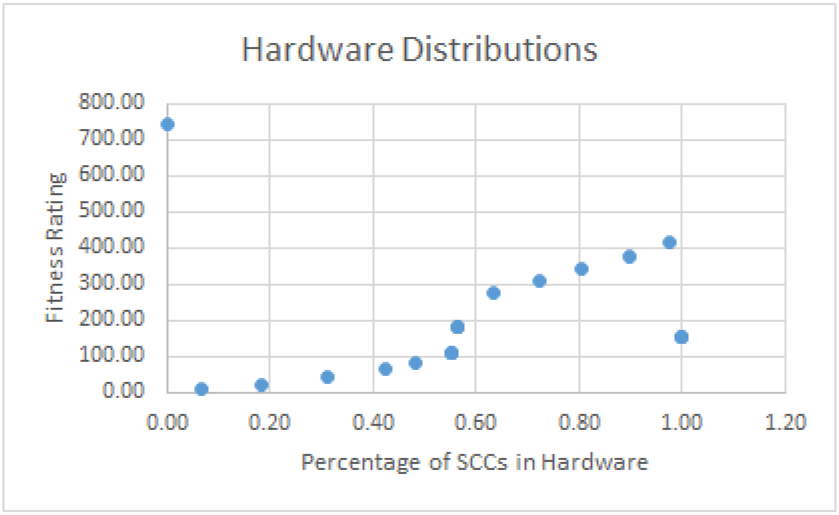


Figure 35 MPEG-2 Fitness Ratings

Other test code shows performance favoring high hardware counts up to a point where higher hardware counts do not result in lucrative trade offs. Fitness ratings of the same outcome show extreme favoritism to low hardware counts because of its implementation of SW_COST and HW_COST. Unlike performance ratings, fitness ratings only count SW_COST once. Leveraging both, the best spot for minimal ratings is at 50%.

7.3 Tabu Search Simulated Annealing Results

7.3.1 Simulated Annealing Neighborhood Generator

Creating SANG inside LLVM required the creation of a cost model, and it is built into DSWP's pass as the "Partitioning Algorithm". This made it so that SA could be run at any time on a given PDG. Inside a graph node, trait1 and trait2 dictated the starting heat and cool down increments respectively. The heat of a node was decided to factor into the A/B calculations to give a contour to the energeticness of a system, and the cool down factor was tuned to give a meaningful decrease in the chance for a solution to change drastically. The initial head was also tuned to ensure that the initial runs were radical enough that the chance for a solution change was extremely high.

As noted before, SANG examines the parent solution initially. For each instruction encountered, it checks the parent solutions partition for that instruction. If it exists in the parent's SW partition, it will continue the evaluation. If it is not part of the SW partition (and thus is part of the HW partition), it will turn in back into SW. Now back to the SW instruction, there are two chances it has to become HW. This was done by inverting the partition assignments for each SCC instructions.

Either a random(0,1) variable being greater than exponent(A/B) or A > B will turn an instruction into HW. This A B comparison was decided to be derived from the current latency costs and the total costs respectively. The costs were also normalized. This process of HW/SW partition assignment goes on until all the instructions have been accounted for. After this the performance (TSSA) and fitness (GA) costs can be evaluated.

Tuning was required to give the random(0,1) comparison a "valid" competitor, being exp(A/B). It was desired to have exp(A/B) fall into 0-1.0 so that the random 0.0-1.0 could create a threshold when exp(A/B) gets evaluated. Along with this evaluation, the A > B is also going to be investigated as well, as a balance should be struck there.

The cost model was done using an accumulating delay time (for the code being processed) called SW_TIME while there also existed the HW_TIME result for a given instruction. Like originally, the delay time is extracted from an estimation on the latency cost for the given instruction. This latency cost can be used to represent SW_TIME, being the time cost of running the system in software. Along with SW_TIME, HW_TIME takes the latency and divides it b a scalar value to simulate the speedup of having an operation in silicon. Along with the general time costs, there is also the "generic" HW_COST/SW_COST. This cost value is supposed to represent a second constraint, such as area or power. The random element in SANG, and the variability of trait1 and trait2 allowed some difference in the distributions of hardware and software.

To show the operation of the algorithm, we have here a parent partition distribution. The instructions are ordered in an array by the order of occurrence during the SANG pass. As explained, it's clear that the highlighted points are the ones that can be turned into HW since they exist as SW in the parent. Anything assigned to HW will become SW automatically. This operation is carried out many times, for a set number of generations. With each generation of the SANG pass, in full implementation, the solution set grows drastically. Assuming 5 children per parent are created, we get the following growth in Table 2.

Table 2 Solution Node Growth

Iteration	Explored Nodes	Total Nodes	Children Created
1	0	1	5
2	1	6	30
3	7	36	180
4	43	216	1080
5	259	1296	6480
6	1555	7776	38880
7	9331	46656	233280
8	55987	279936	1399680
9	335923	1679616	8398080
10	2015539	10077696	50388480

Iterations are the amount of times that the edge nodes (children in the previous pass) as explored and used to generate new children. Explored nodes is the total amount of nodes that have had partitions generated and rated. Total nodes are the existing nodes at the time of the given iteration (this includes the unexplored fringe nodes). Children Created as the results of the fringe nodes generating five children each. As it can be seen, there is an exponential increase in the amount of nodes required for each consecutive SANG pass.

Assuming we iterate 6 times: in order to run the next iteration, there must be all of the total nodes generated at a given time, which begins to rise drastically by iteration 7. With 233,280 nodes created, each having a map of HW/SW partitions, while keeping track of 46,656 current nodes, a typical machine will start having issues running the search any further. Given sufficient time, memory, and processing power though, an exhaustive SANG search can probably reach a sufficient solution.

One major limitation is memory, as assuming each node requires 1 kilobyte of memory, then by iteration 7 we require 233 megabytes of memory to observe each node. After this point, further iterations reach the gigabytes, with iteration 10 requiring 10 gigabytes. The amount of paging and swapping necessary to render this correctly becomes ludicrous, and a local search shaping algorithm starts to sound extremely refreshing.

7.3.2 Cuts and Hardware Nodes

It was desired to generate data based on the amount of SCC instructions assigned to hardware and the cuts done over the PDG of SCC instructions. This data was collected over a variety of test cases and examined to ensure that the number of cuts and hardware instructions were not linear, since the same number of cuts done in different ways can generate different amounts of HW/SW distributions. If this relation was extremely linear, this would indicate a problem in the SANG system as one of the hinging factors is its ability to do widely different cuts than Twill's original Accumulator method. A general trend of more instructions/more cuts was expected still though, as the more variety there is in a system, the more the system will have the irregular element. The irregular element is hardware in this case, as the system starts of originally as all software. The results are shown in Figures 36-38.

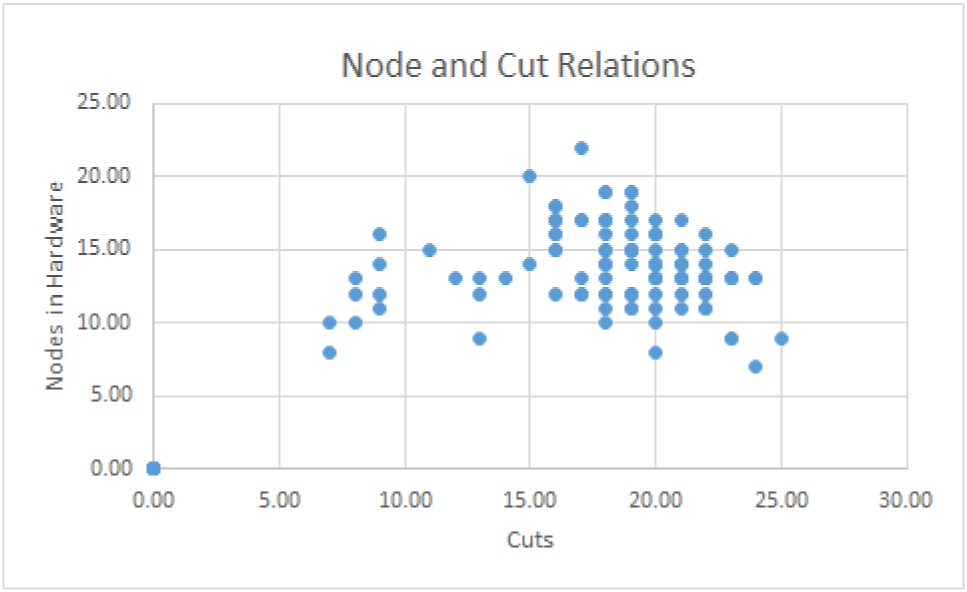


Figure 36 MIPS Cut and Node Relations Using SANG + Performance Ratings

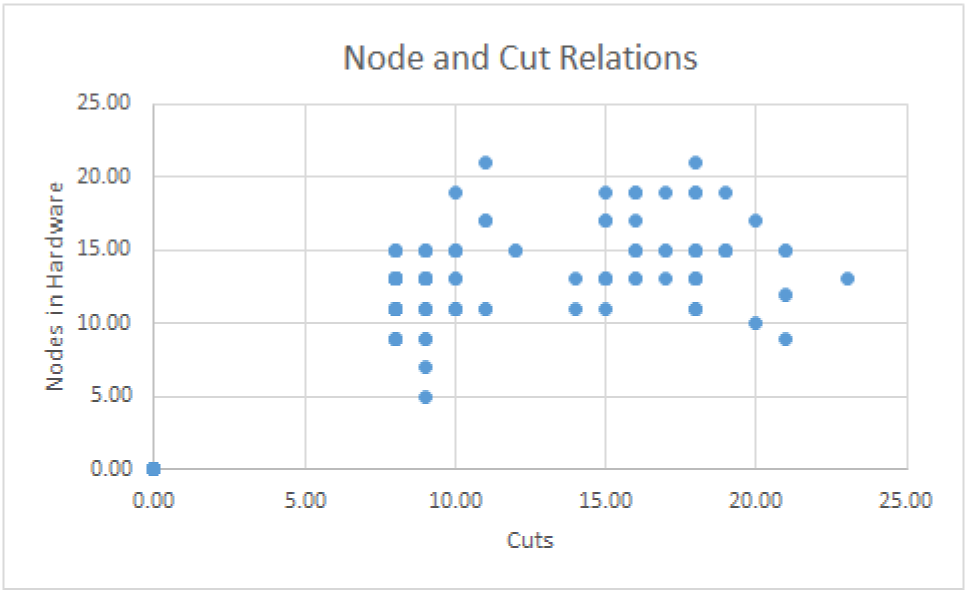


Figure 37 MIPS Cut and Node Relations Using SANG + Fitness Ratings

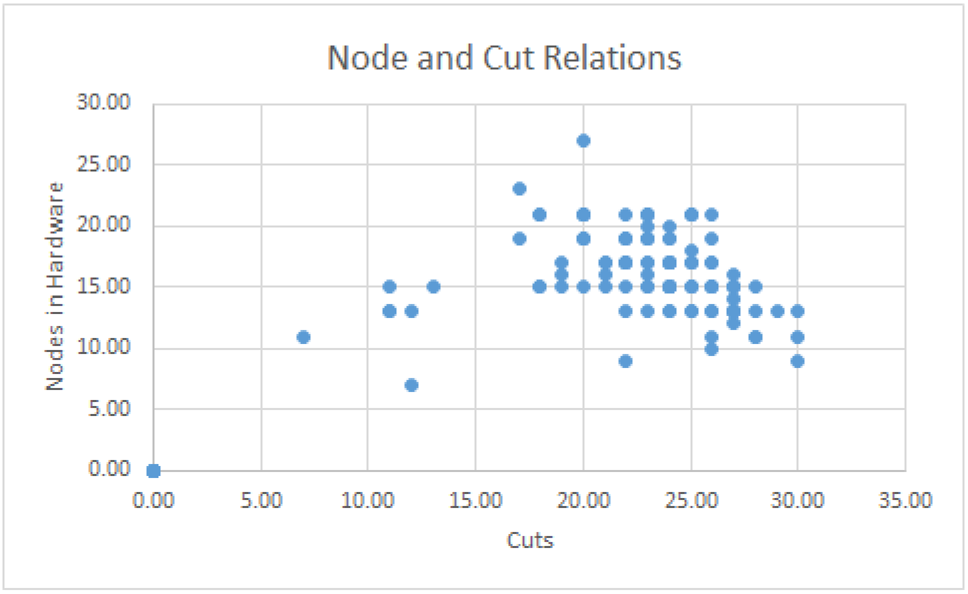


Figure 38 Blowfish Cut and Node Relations Using SANG + Performance Ratings

The cut and hardware distributions were expected, and the spread of cuts and hardware instructions shows that SANG has a very broad search relative to both the cuts made and the amount of hardware used. This is important, as this kind of variability is desired in a search like Simulated Annealing.

7.3.3 Cuts and Scores

The cuts that were made and the scores that are achieved by these algorithms is one of the main concerns. To gather data on the score/cut tradeoffs, iterations of the SANG algorithm were run using different starting temperatures, cool down rates, and depths. These were modulated and then scored against each other to see what configuration the algorithm should be in to get the strongest result. The algorithms had cut counts and scoring added to them in order to make this data gathering possible. The cut and HW instruction distributions were tallied in a variety of test cases, using code such as a MIPS simulator, AES encryption, or a Blowfish Encryption.

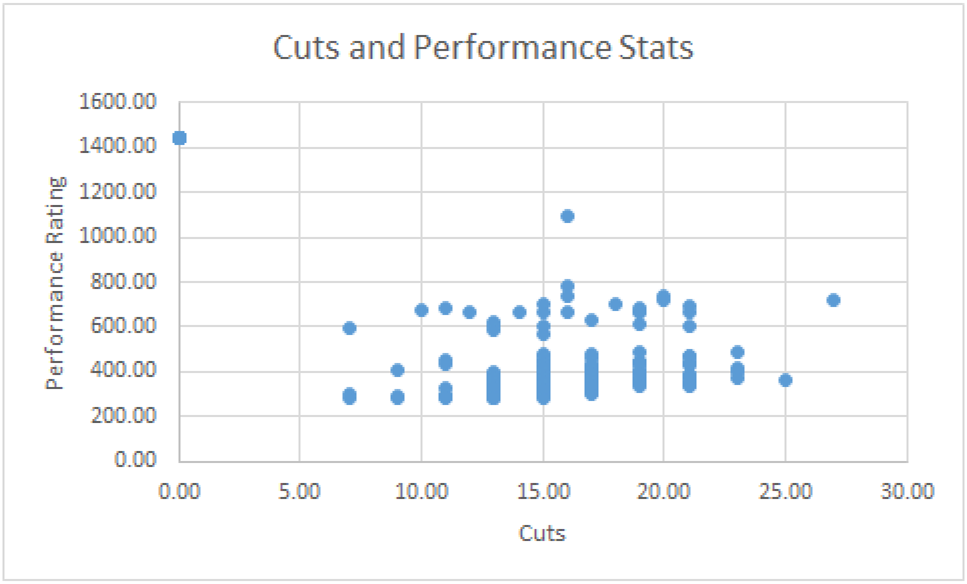


Figure 39 Blowfish SANG Performance by Cuts Results

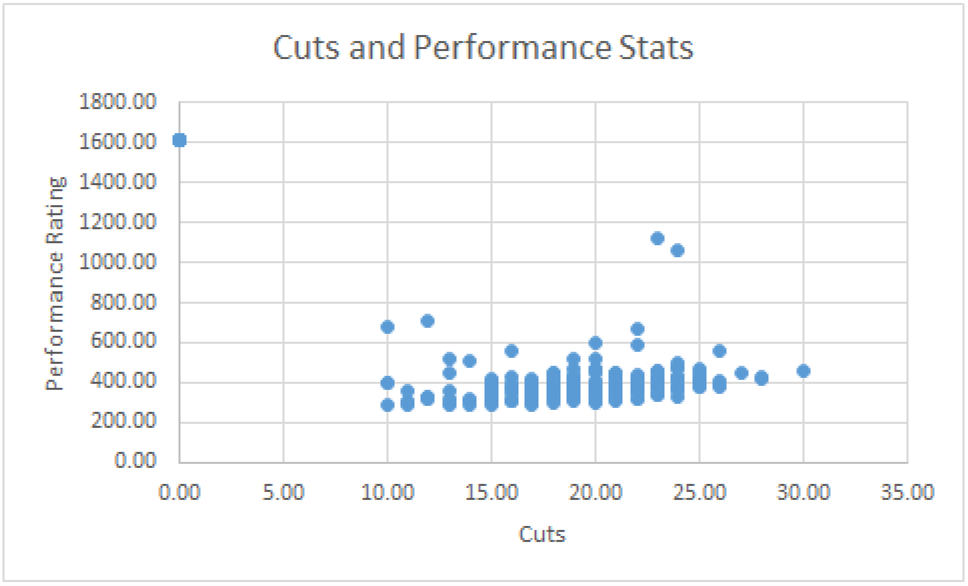


Figure 40 MPEG-2 SANG Performance by Cuts Results

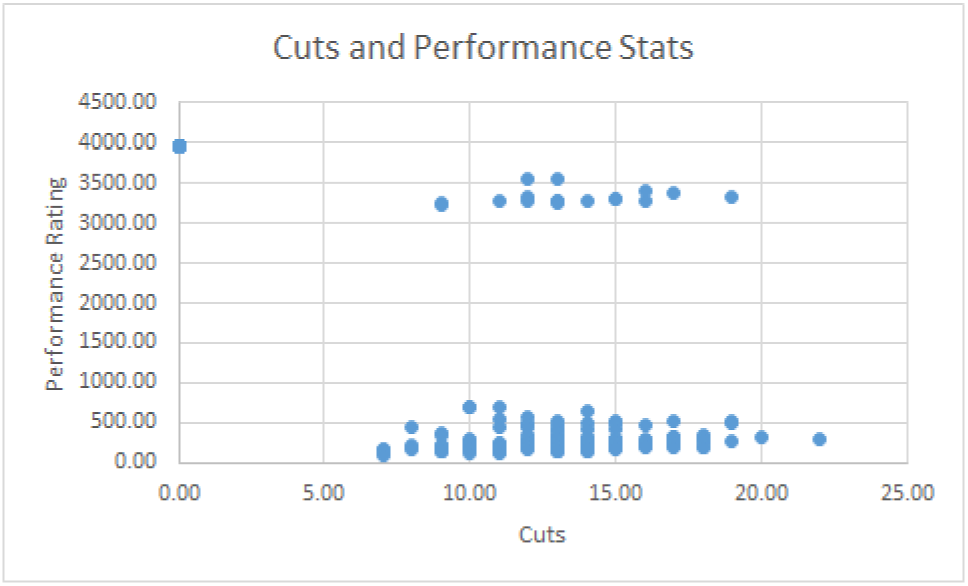


Figure 41 MIPS SANG Performance by Cuts Results

It's clear that a wide range of cuts can get you some similar performance ratings, seen in Figures 39-41. The lowest ratings show up with the lowest amounts of cuts, meaning that a careful cut method will garner a better performance score as opposed to a large amount of cuts, such as 25~30. With more cuts comes more queues, and more communication delays. The Performance Rating system does take into account these ratings. To preemptively compare TSSA/SANG with the genetic algorithm, the genetic fitness scoring will be examined as well. Firstly, with SANG as the shaping and Performance as the Scoring, the MIPS code sample got the results shown in Figure 41. Scoring the solutions found by SANG/Performance scoring using the Fitness scoring showed the following cut/fitness distribution in Figure 42.

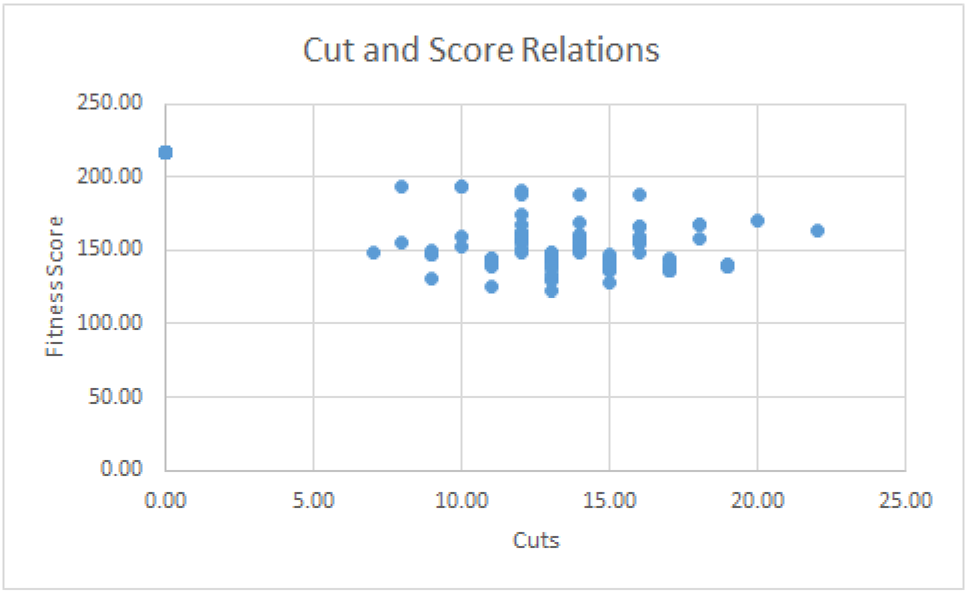


Figure 42 MIPS SANG Fitness by Cuts Results

While in the Performance Algorithm there is a split, showing that there are optimal and no optimal cuts, with the fitness algorithm and it's disregard for communication costs, there are no optimal or non-optimal cuts.

Changing the system to run a SANG/Fitness scoring was then done to study the effects of using the fitness algorithm. Both the performance and Fitness Ratings were seen. This configuration means that SANG was still used, but the fitness rating was used to contour the results. The following Figures, 43 and 44, uses MIPS code as the input.

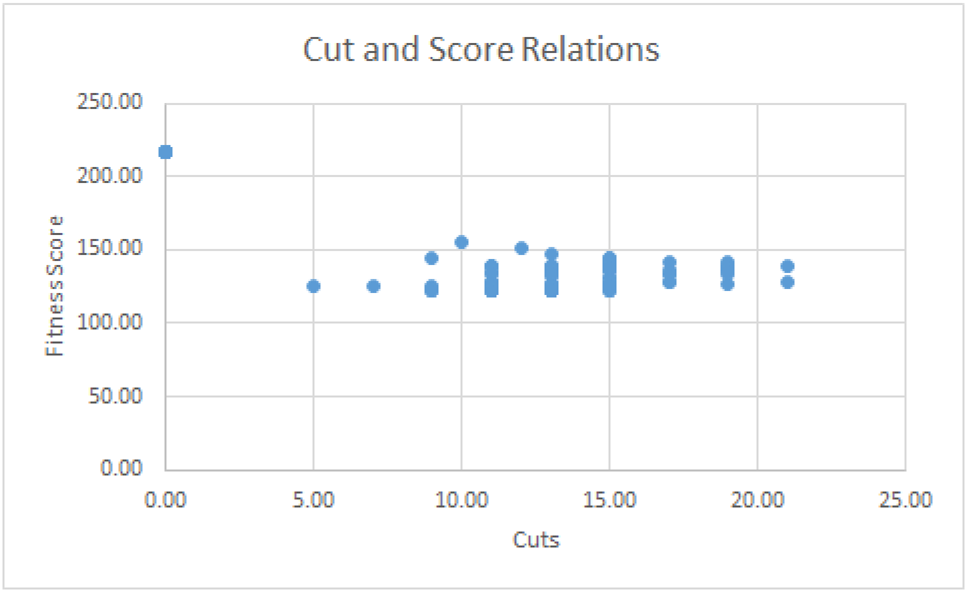


Figure 43 MIPS SANG Performance by Cuts Results

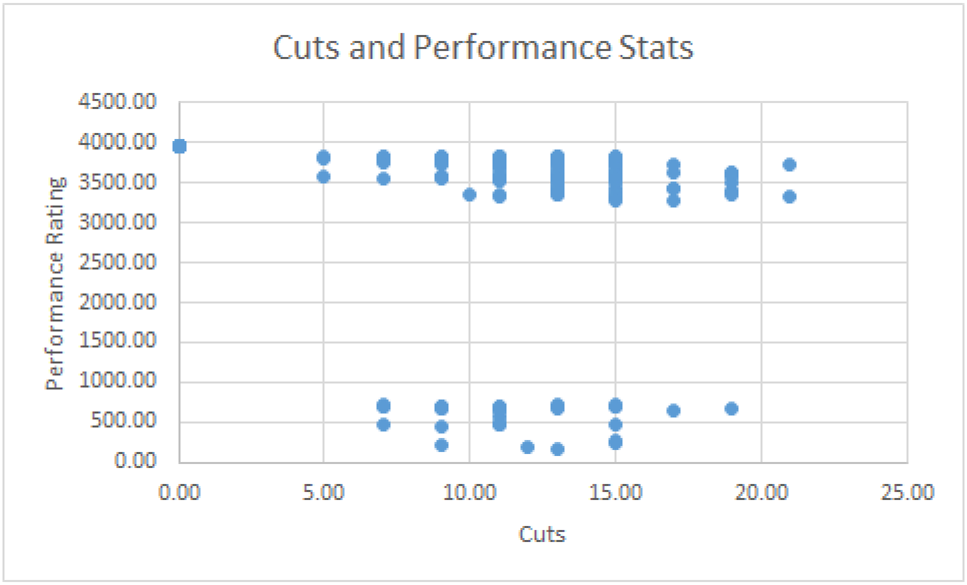


Figure 44 MIPS SANG Performance by Cuts Results

The performance scores rise dramatically, and the fitness scores do not improve compared to the SANG/Performance scoring, which garnered the same Cut and Score results with admittedly more spread, but as a whole using the SANG/Fitness combination of scoring harmed the Performance Rating of the solutions (105 compared to 169).

7.3.4 Hardware and Scores

While the cuts and score relations give valuable data, the relation of hardware instructions to score is also desired. Shown from both the data seen in cuts made vs. hardware instructions data and cuts made vs. performance scoring, we can expect that we will have a range of performance outcomes for the same number of hardware instructions.

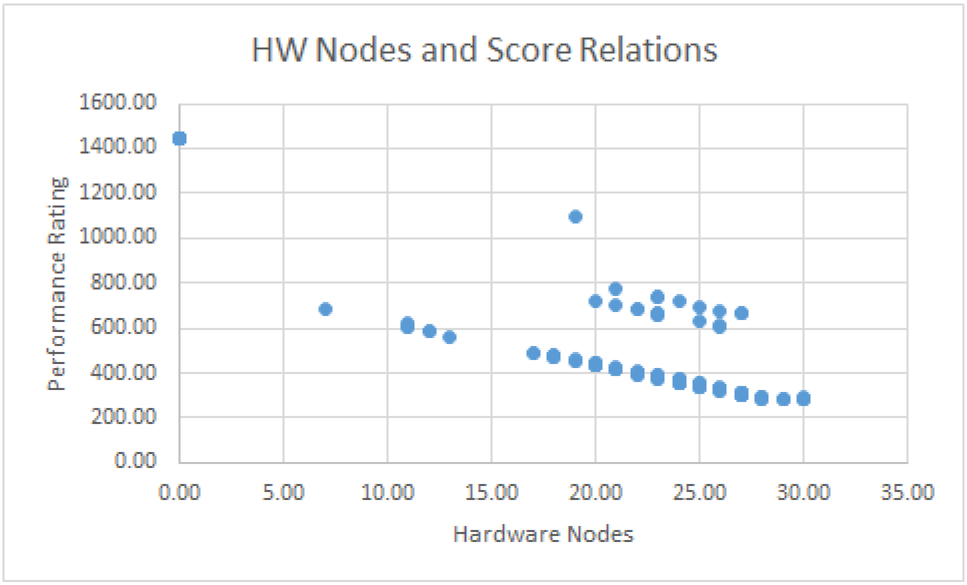


Figure 45 Blowfish HW Nodes Used with Resulting Performance Ratings

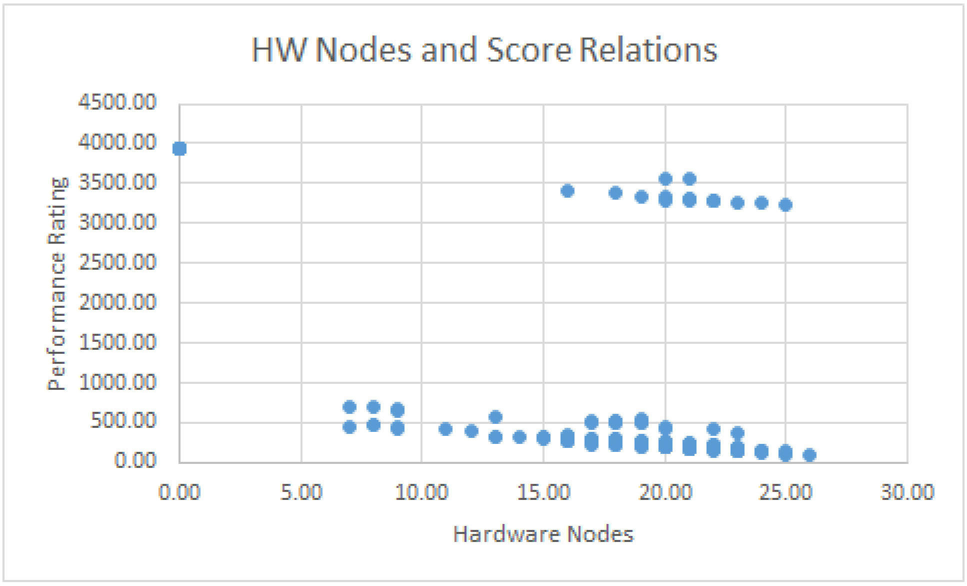


Figure 46 MIPS HW Nodes Used with Resulting Performance Ratings

A general downward trend is observed in Figures 45-46, but with a range of instructions in hardware, a split is seen in the rating while the hardware instruction counts stay the same. The cost differences can be quite drastic, as shown in the MIPS result as opposed to the Blowfish result.

7.3.5 Iterations and Scoring

To ensure that iterations did indeed shape the Performance Rating to an optimal value, the scores found were plotted with the iterations they were found with. This is done to show the range of solution outcomes that can be expected with a given set of iterations seen in Figure 47.

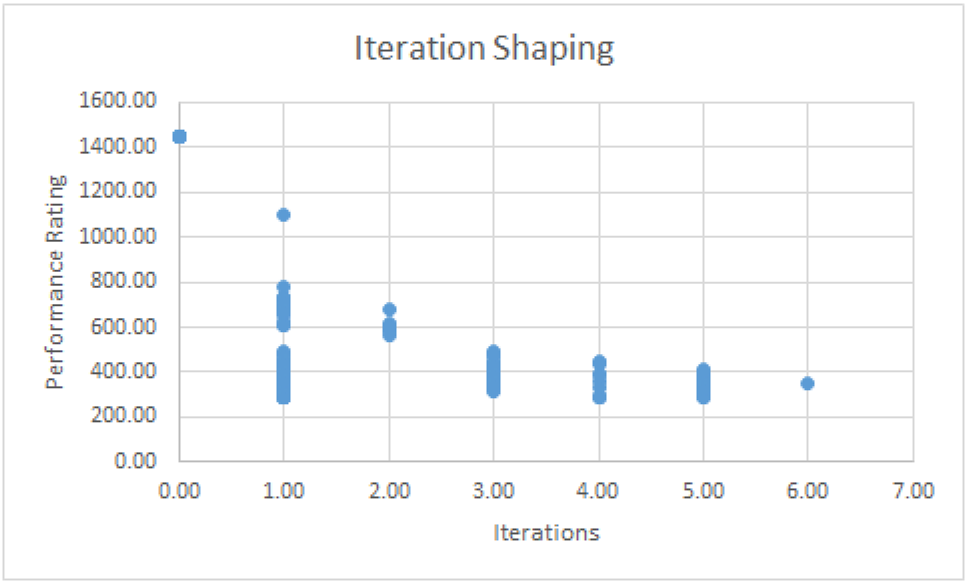


Figure 47 SANG Iterations with Resulting Performance Ratings

As the iterations increased, the score range for best results shrank exponentially as the ideal outcome was found. There is a tradeoff, as described beforehand the amount of memory and time needed to operate on higher iterations is massive, and tabu search will help localize this search, allowing deeper iterations without having to use massive amounts of processor time and memory to generate a solution. It is clear that these graphs all follow a very similar distributions. This means that despite the randomness, SA was still shaping the outcome of our partitions according to a cost contour defined by the A/B definitions (used in the comparison and exponent calculations). While any of solutions found with SANG are valid, let's see if we can get tighter distributions around the best case scenario using the tabu search.

7.3.6 Tabu Search

The tabu search will act as if it can only see locally and move through the SANG map we have created. Unlike the A/B focused SANG pass, the Tabu Search only cares about comparing performance of each algorithm. This means that the more complex performance calculation is what defines the tabu search. Starting at the center solution node, it examines it and all its children, looking for the best solution, as noted before with the idea of the "TAU" solution node. Upon exhausting the nodes, it sets the tau node as tabu and gathers the next set of children and continues its search. If a tau node was found to be gamma (the best possible so far encountered), then it will be set as gamma.

Assuming that the TSSA algorithm only creates SANG children when needed we get the following solution node requirements per iteration in Table 3.

Table 3 Tabu Search Children Growth

Iteration	Explored Nodes	Total Nodes	Children Created
1	1	6	5
2	7	11	5
3	18	16	5
4	34	21	5
5	55	26	5
6	81	31	5
7	112	36	5
8	148	41	5
9	189	46	5
10	235	51	5

Unlike an exhaustive SANG pass, we do not need to save the parent nodes after creation, but even with saving, after 10 iterations we cover 235 nodes rather than 2015539, in the time it takes for roughly 5 generations of SANG to be created in full. Since TSSA can explore up to 10 iterations deep in this time as shown in Table 3, it has the chance to find solutions in iterations 6-10 of SANG that would have taken much longer to generate. This directed and greedy search that follows the best performance node contours and shapes the outcome distribution of HW/SW nodes aggressively. We can see this search actively running with the following Performance vs. Iterations graph for various input samples.

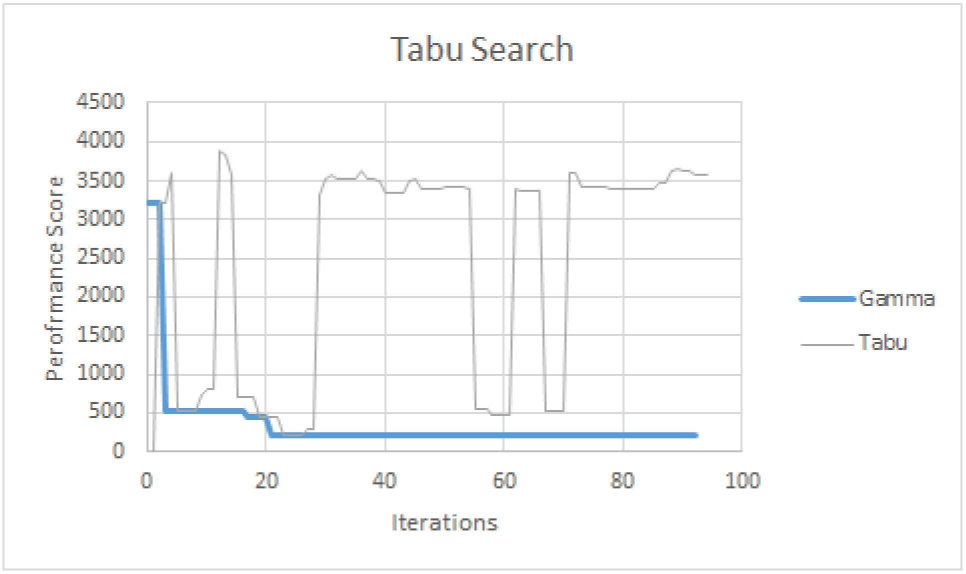


Figure 48 MIPS Code with the TSSA Pass

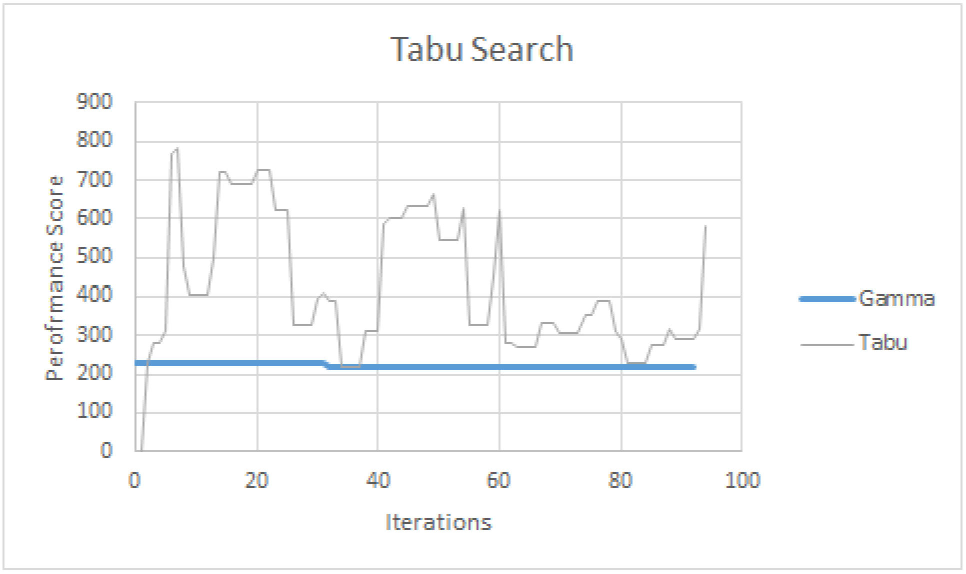


Figure 49 GMS Code with the TSSA Pass

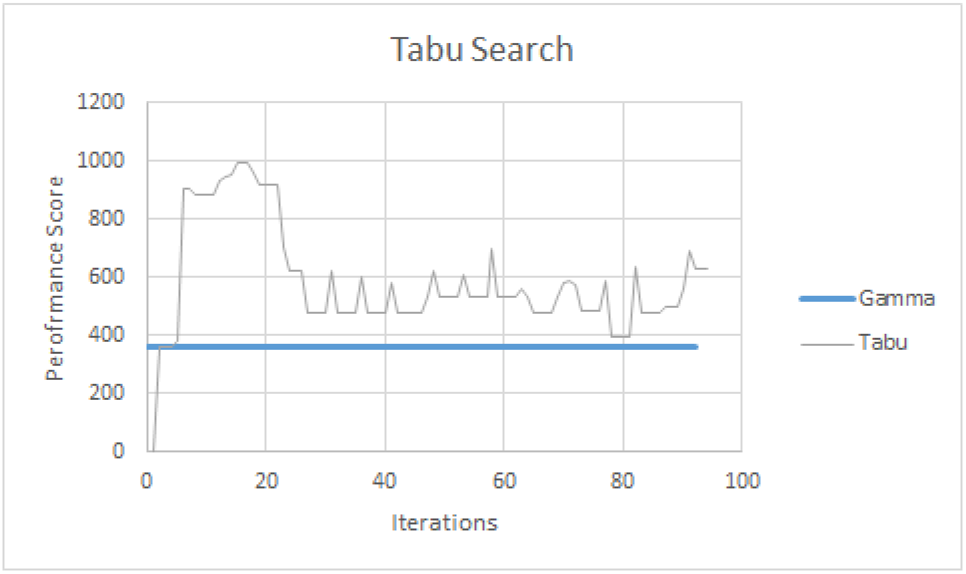


Figure 50 AES Code with the TSSA Pass

As it can be seen in Figures 48-50, the tabu node in each iteration is shown alongside the gamma node. The gamma node is the best one found so far. Sometimes the Tabu Search does get more concrete results than the SANG passes that it searches across, but if early distributions of HW/SW SCC instructions are effective enough they will stay as the gamma node if no better solutions can be found, like in the AES example.

7.4 Genetic Search Results

7.4.1 Population Evaluation

For each generation, the population is defined initially by just a genome and then is generated into an actual partition distribution. This distribution includes the ratings for fitness and performance, just like the TSSA/SANG/Random/Allocator solutions. After the solutions are evaluated, their genomes will be extracted, and the solutions will be destroyed (unless it is the best possible found). This allows the genetic search to stay local, and avoid rapid exponential growth like the exhaustive SA algorithm.

After creation, the population is sorted by fitness to create a distribution with the highest performing solution on one end of the spectrum and the worst on the other end of the spectrum. Once this occurs, two random numbers are rolled to determine which solutions to pick out of this sorted array of solutions.

7.4.2 Crossover and Mutation

Once two solutions are selected, their genomes are crossed over by choosing a split point in the alleles and copying over the needed data into two new genomes. After crossing over, a modulo operator is called with respect to the size of the genome array. For LIMIT_MUTATION times, the bit designated by the modulo operator is flipped. Since we usually are dealing with a SW/HW partition, software becomes hardware and vice versa.

After mutation is over, the process starts again with new genomes.

7.4.3 Cuts and Hardware Instruction Nodes

Like in the SANG algorithm, the amount of instructions compared to cuts was studied.

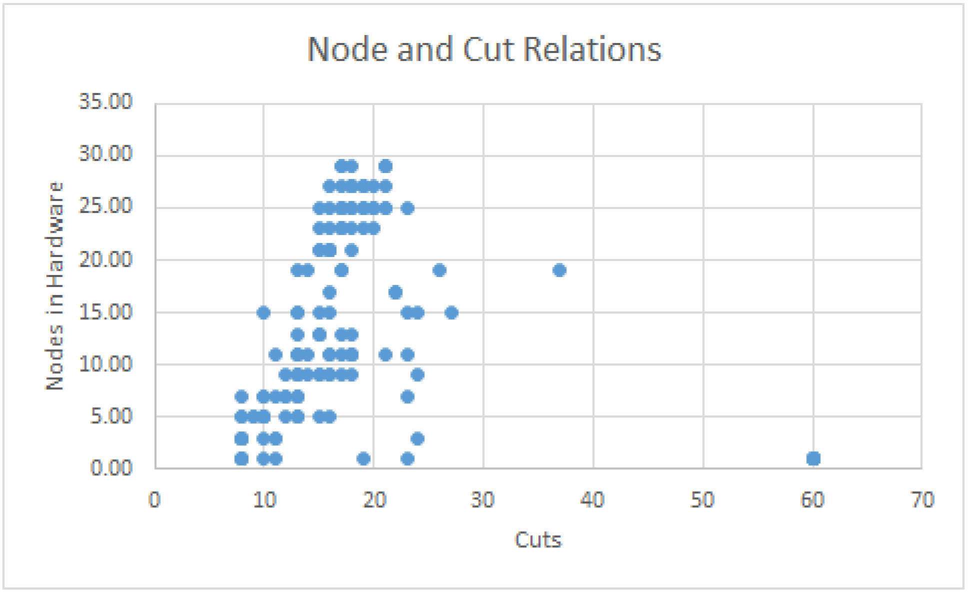


Figure 51 MIPS Nodes and Cuts Relationships

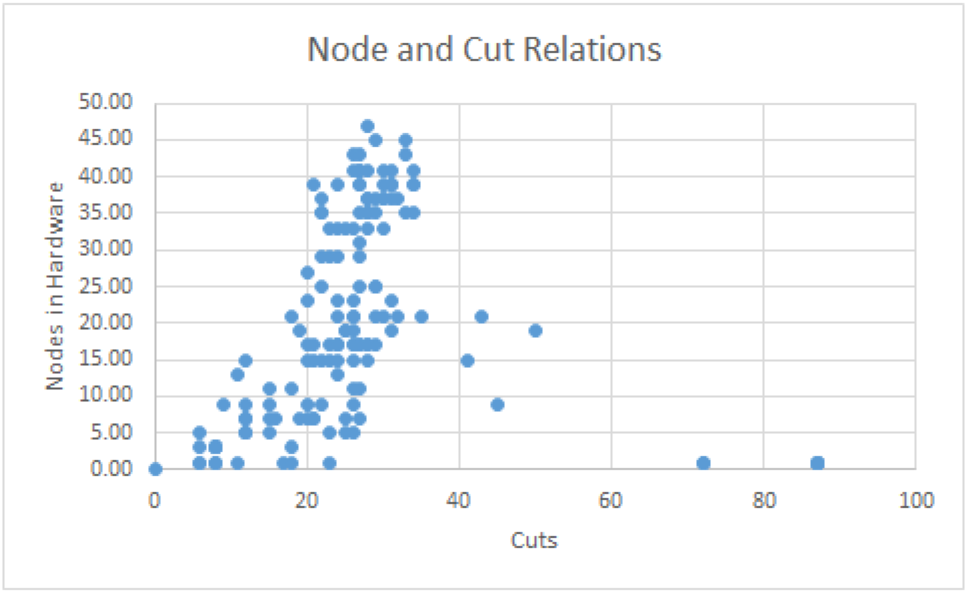


Figure 52 MPEG-2 Nodes and Cuts Relationships

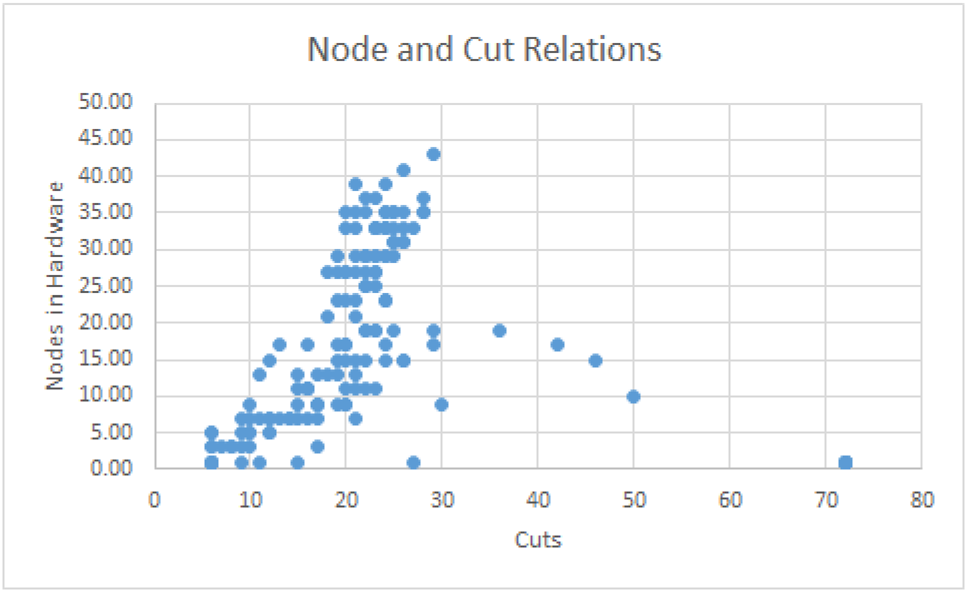


Figure 53 Blowfish Nodes and Cuts Relationships

All of these distributions in Figures 51-53 were created by Genetic Algorithms using Fitness as a rating system. The characteristics of each graph are similar to one another, showing a linear increase from 10-30 cuts, and a curve downward past that point. Some of the higher cut solutions had extremely low numbers of hardware in them, meaning that hardware was becoming "pockmarked" with small bits of hardware interleaved with software.

7.4.4 Cuts and Scores

The relationship between the number of cuts and the fitness score calculated was also decided.

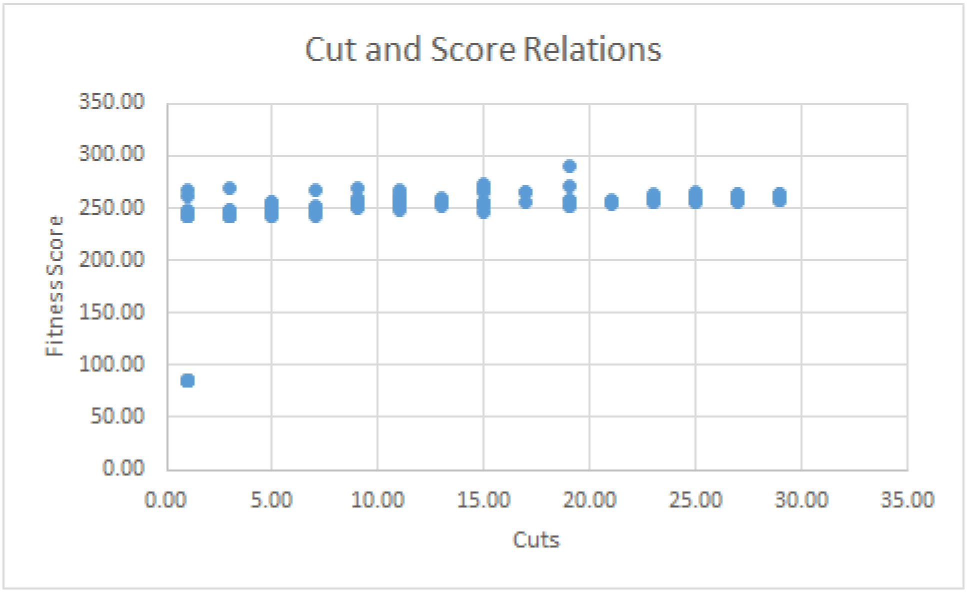


Figure 54 MIPS Fitness and Cuts Relationships

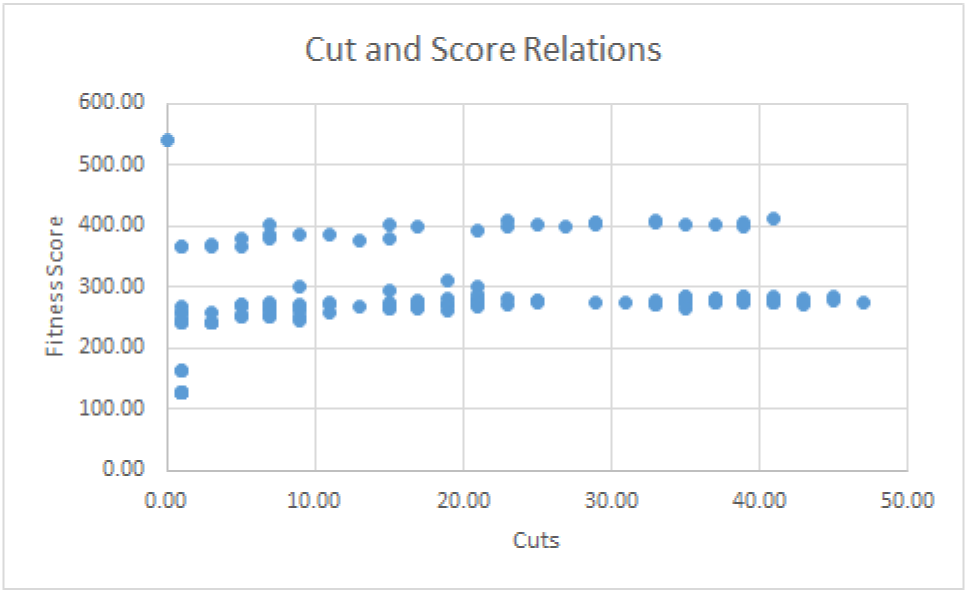


Figure 55 MPEG-2 Fitness and Cuts Relationships

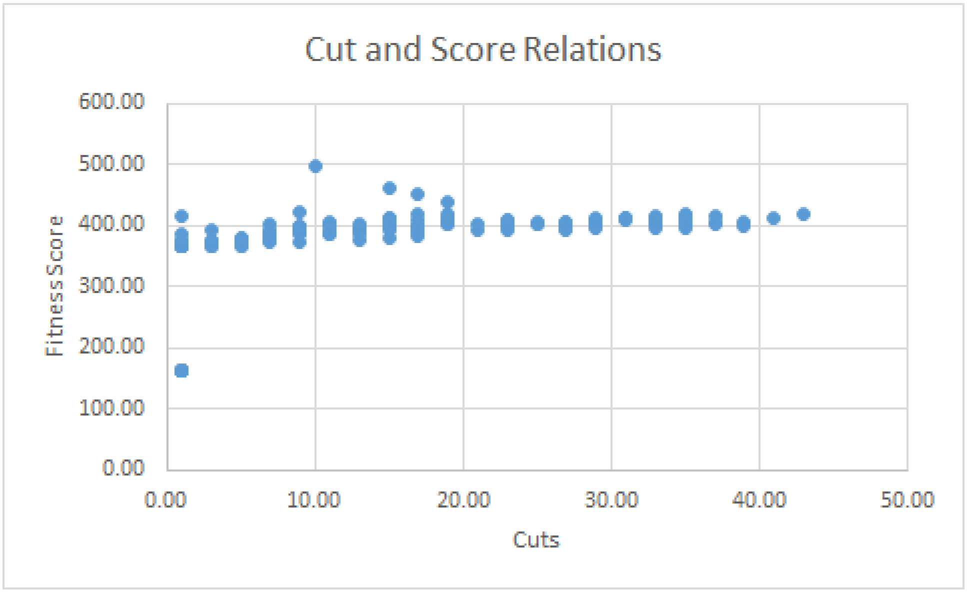


Figure 56 Blowfish Fitness and Cuts Relationships

As shown in Figures 54-56, the fitness score of the genetic algorithm stayed quite stable with the amount of cuts given. It must be kept in mind that the genetic algorithm's fitness calculation does not take communication costs into effect. However, the genetic algorithm's fitness rating does tend to rate less cut graphs with a better fitness score.

7.4.5 Hardware and Scores

Taking a look at the relation of hardware instructions assigned compared with the fitness scores we get the following outcomes.

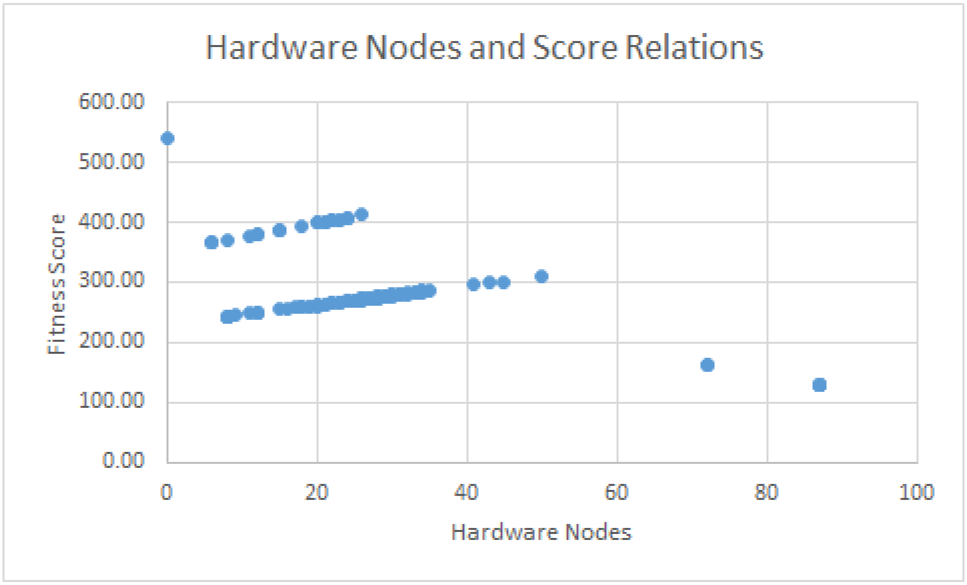


Figure 57 MPEG-2 Nodes and Fitness Relationships

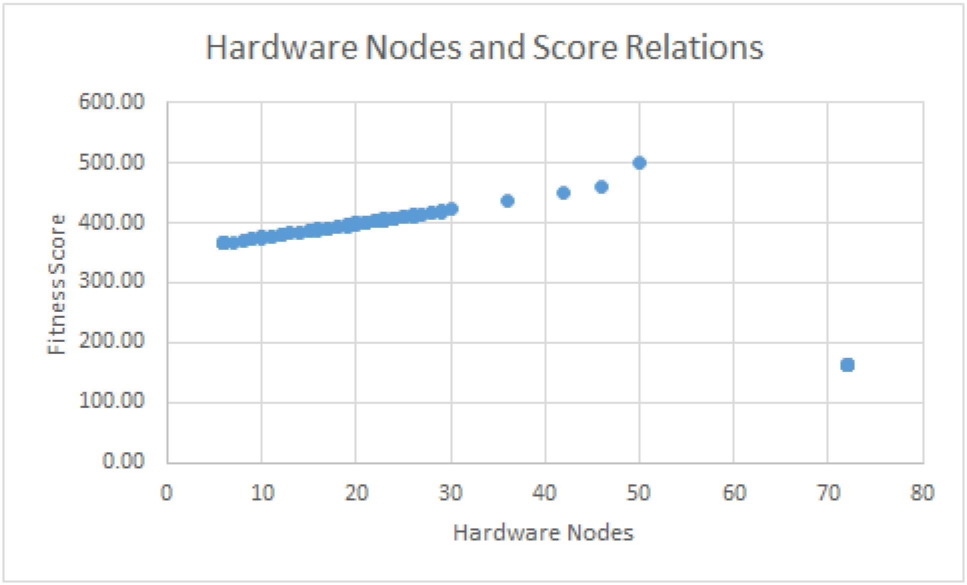


Figure 58 Blowfish Nodes and Fitness Relationships

A lower amount of hardware instructions is seen as lucrative as seen in Figure 57-58, but also a strikingly high amount of hardware is also seen as desirable as well. Each data point is an outcome of a multi-generational pass, and as such, this means that for some population/generation/mutation combination, a high amount of hardware was seen as lucrative for the same problem set that many earlier iterations of population/generation/mutation saw as undesirable. It is also possible that lower population/generation counts would not allow the genetic algorithm to explore the solution space very well, because coupled with lower mutation counts, a software heavy solution may seem like the best.

7.4.6 Time Taken and Score

Along with the general score, it was desired to see if spending more time did indeed generate tighter results. If not, this may mean that the genetic algorithm is not following contours correctly.

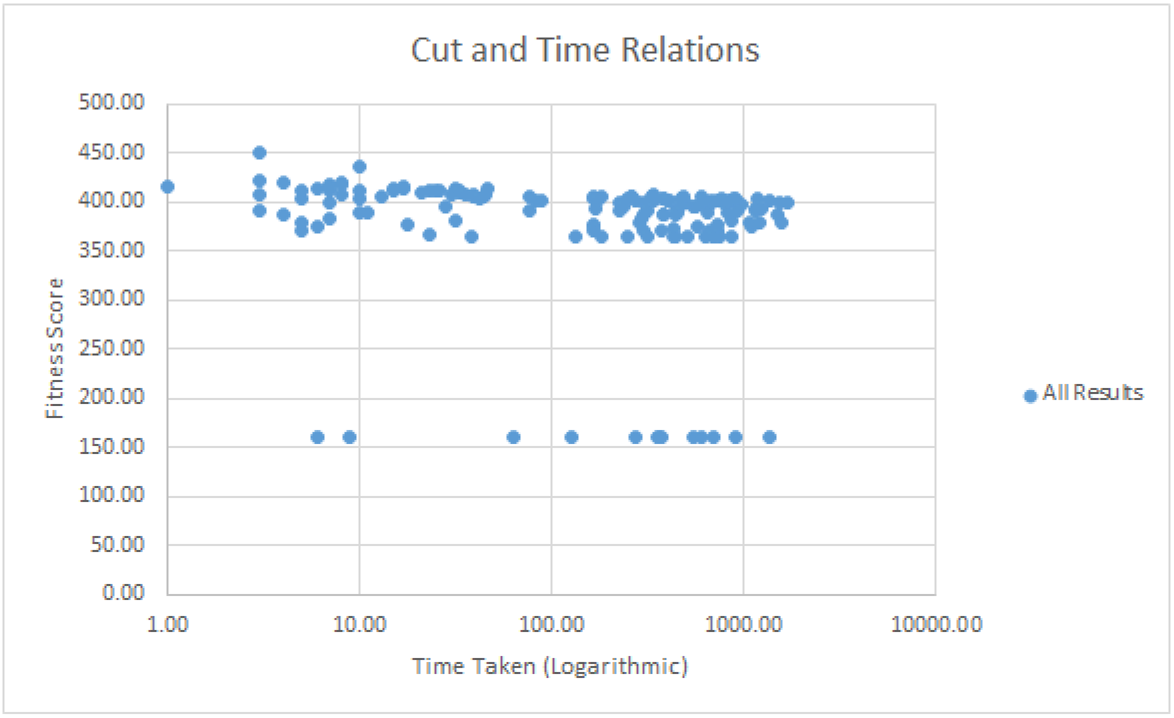


Figure 59 MIPS Time Taken and Resulting Solutions Given Multiple Genetic Runs

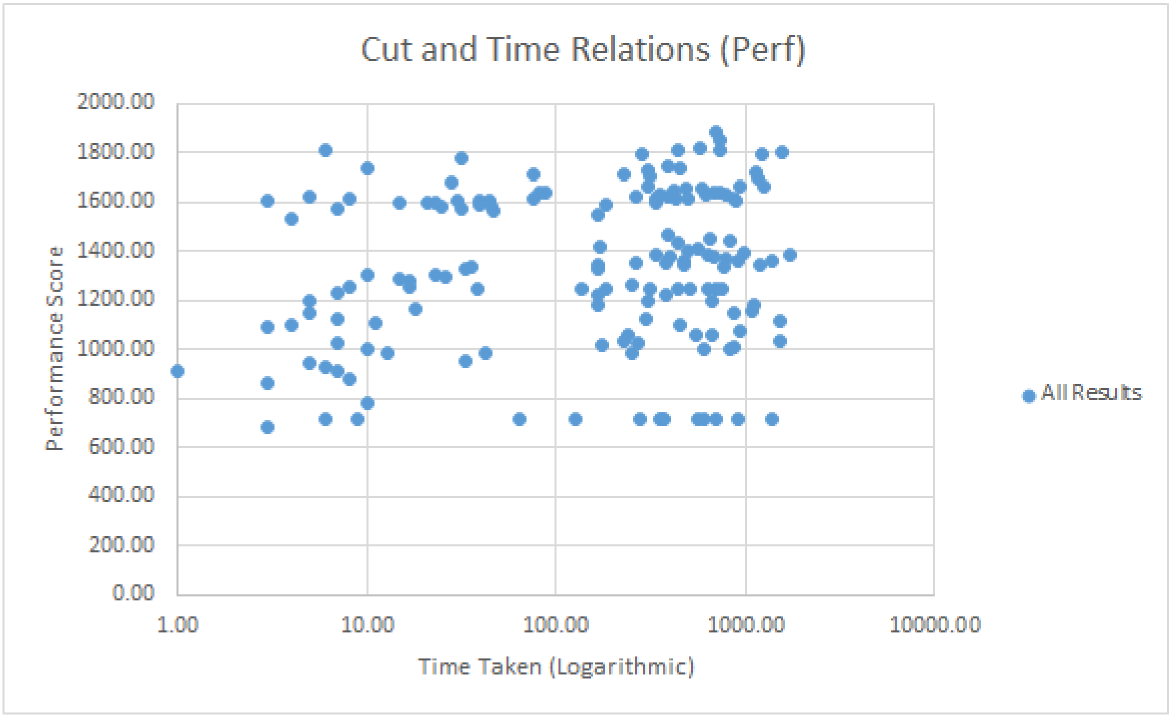


Figure 60 MIPS Time Taken and Resulting Solutions Given Multiple Genetic Runs

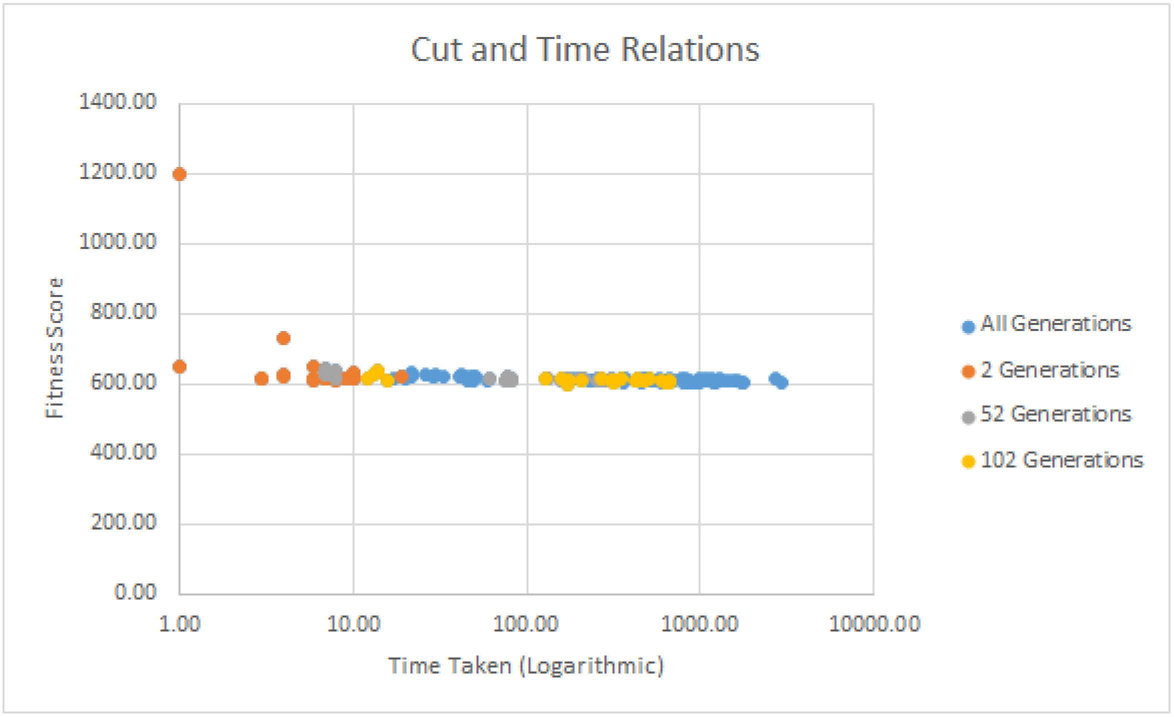


Figure 61 MIPS Time Taken and Resulting Solutions Given Multiple Genetic Runs Using Performance as a Contour

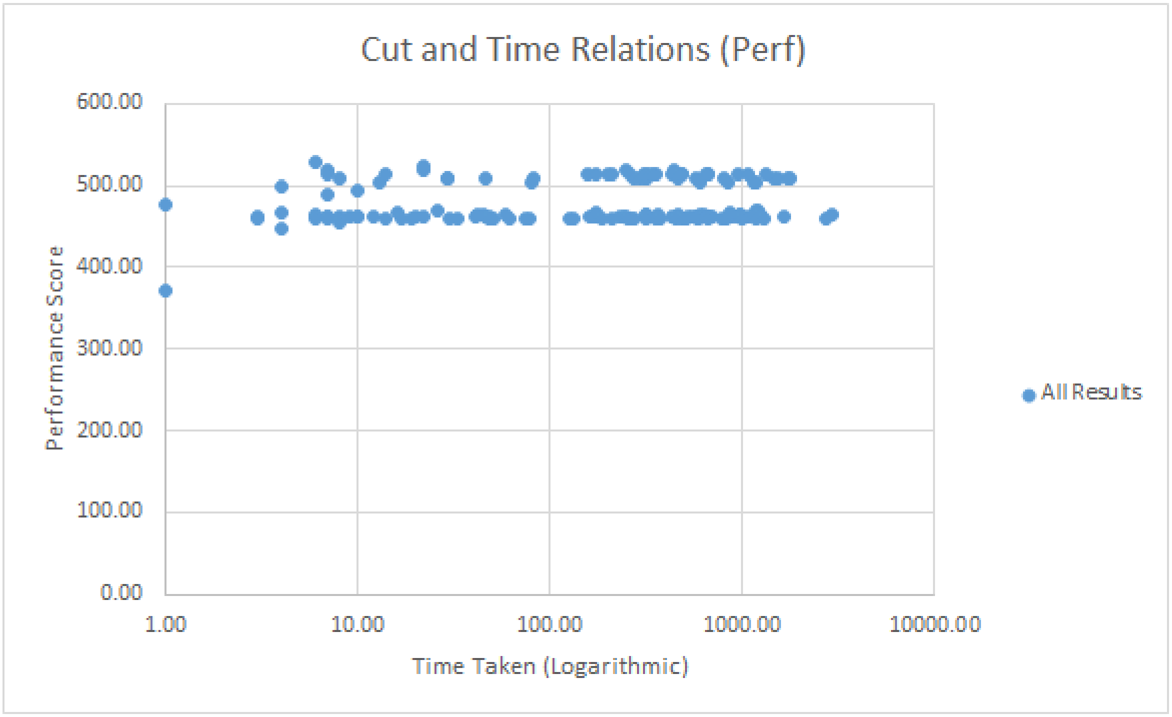


Figure 62 MIPS Time Taken and Resulting Solutions Given Multiple Genetic Runs Using Performance as a Contour

As shown with Figure 59 - 62, using the genetic algorithm together with the fitness rating does get a tighter genetic score as time used to calculate the solution increases. However the performance rating is all over the place, with no clear trend. Using the genetic algorithm with the performance rating instead of the fitness rating does garner tighter performance and fitness scores for the solutions. The changes to the fitness ratings and quite drastic and exponential, while the performance scoring is more gradual, but definitely more spread out at the start as opposed to later.

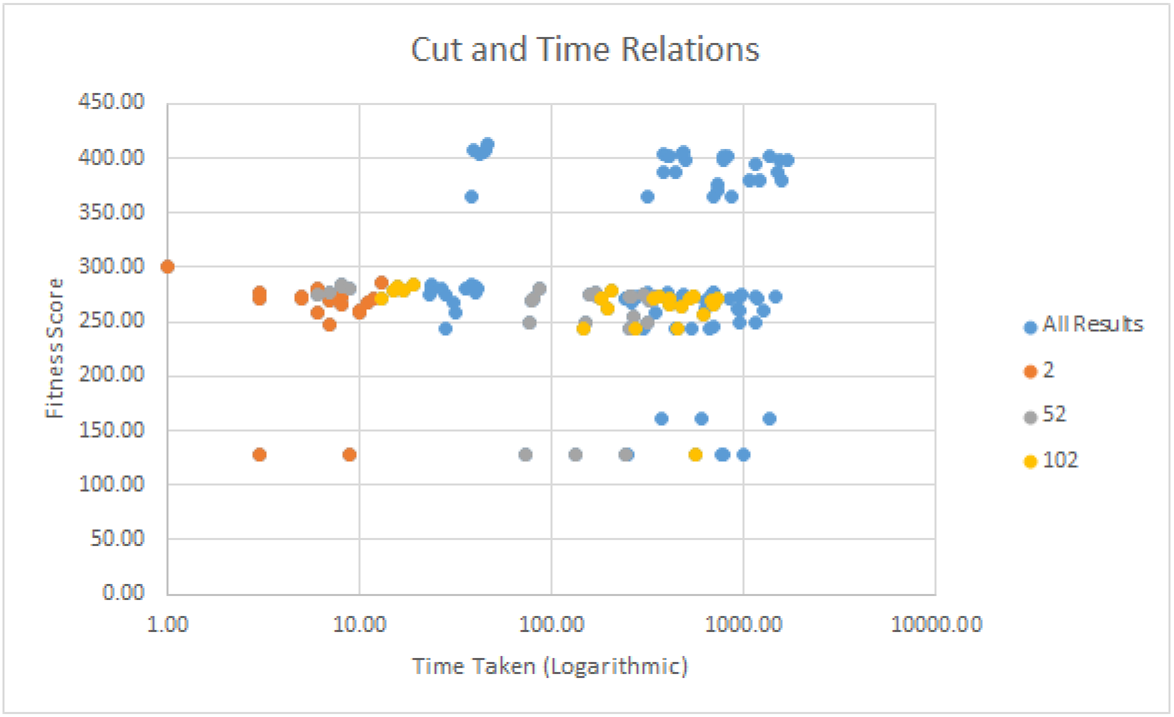


Figure 63 MPEG-2 Time Taken and Resulting Solutions Given Multiple Genetic Runs

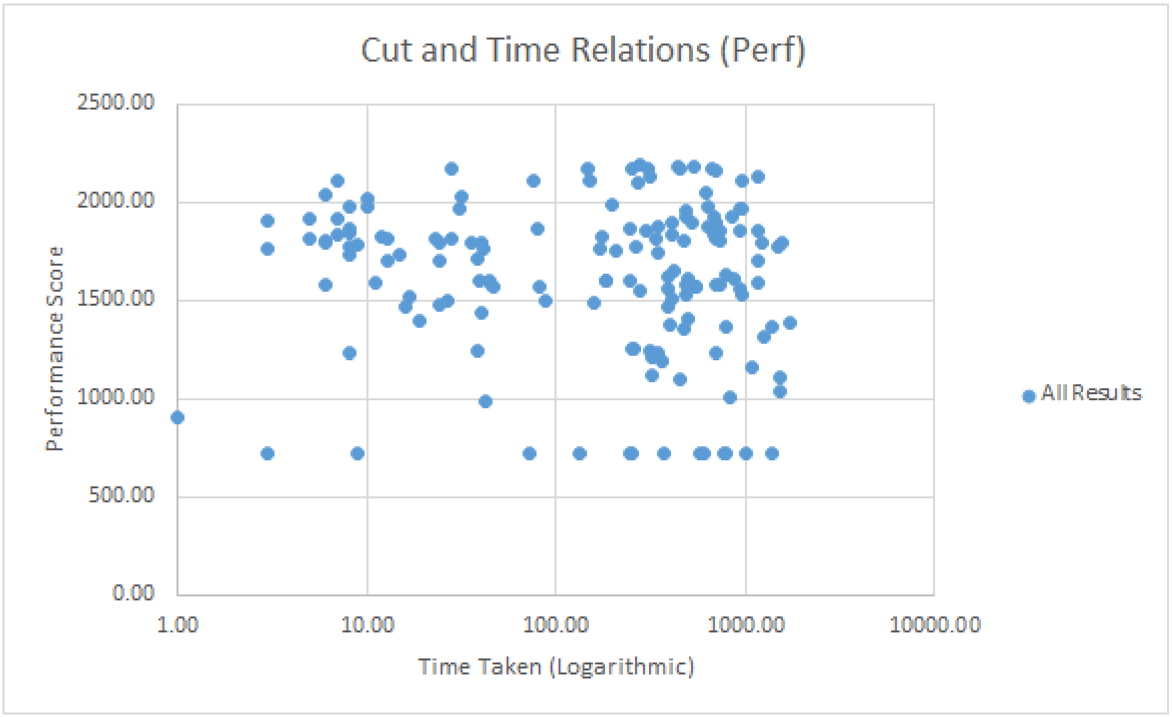


Figure 64 MPEG-2 Time Taken and Resulting Solutions Given Multiple Genetic Runs

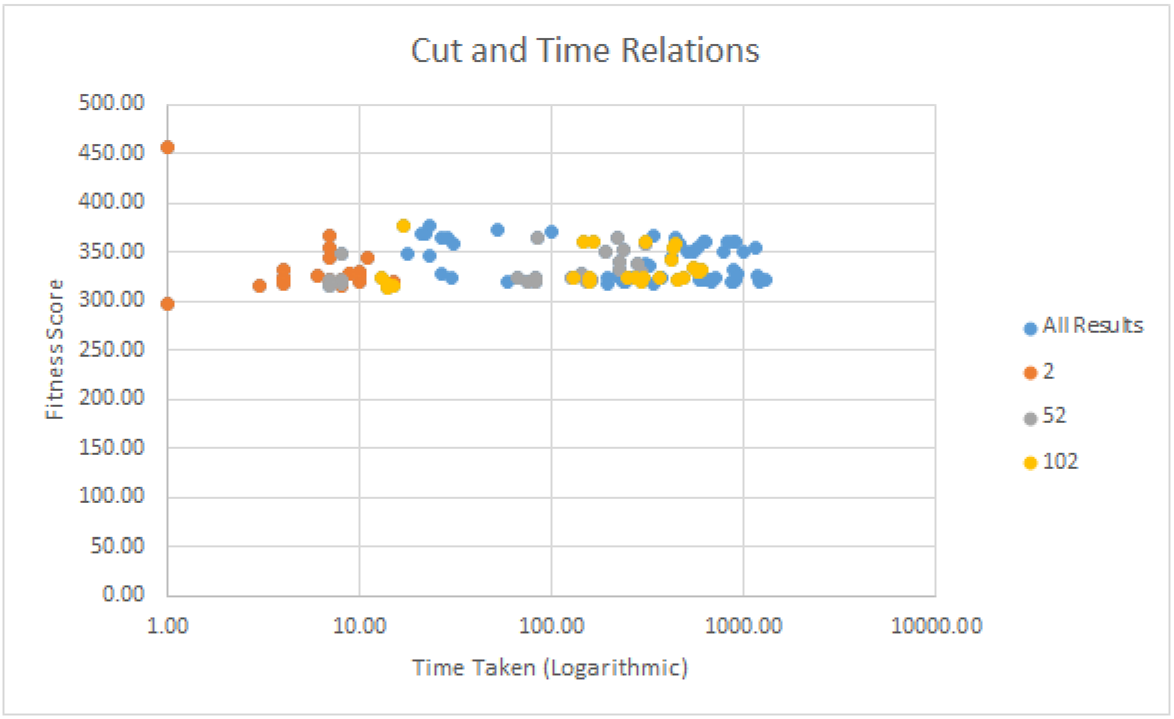


Figure 65 MPEG-2 Time Taken and Resulting Solutions Given Multiple Genetic Runs Using Performance as a Contour

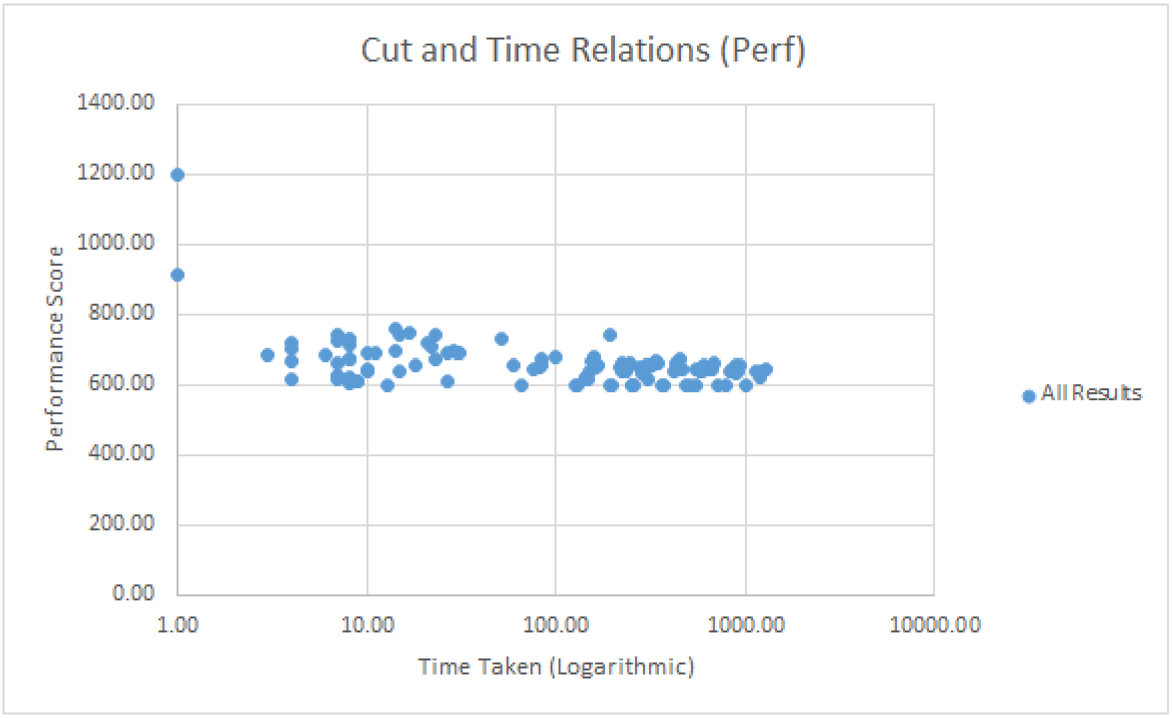


Figure 66 MPEG-2 Time Taken and Resulting Solutions Given Multiple Genetic Runs Using Performance as a Contour

With this test case in Figure 63 through 66, some "bouncing" of scores is also seen. Using the genetic algorithm with a fitness rating, the resultant fitness rating can be in three different places with the performance rating still being scattered. Using the performance rating instead gets a cleaner fitness rating over time result along with a cleaner performance rating over time result.

7.5 Cuts and Queues

While a performance or fitness rating may be good, it was important to see how estimated cuts actually lined up with the amount of queues created. Normalizing the number of cuts and number of queues by the amount of SCC instruction nodes, the following graph was generated using the best case results from Twill's Accumulator algorithm, the Genetic algorithm and the TSSA algorithm.

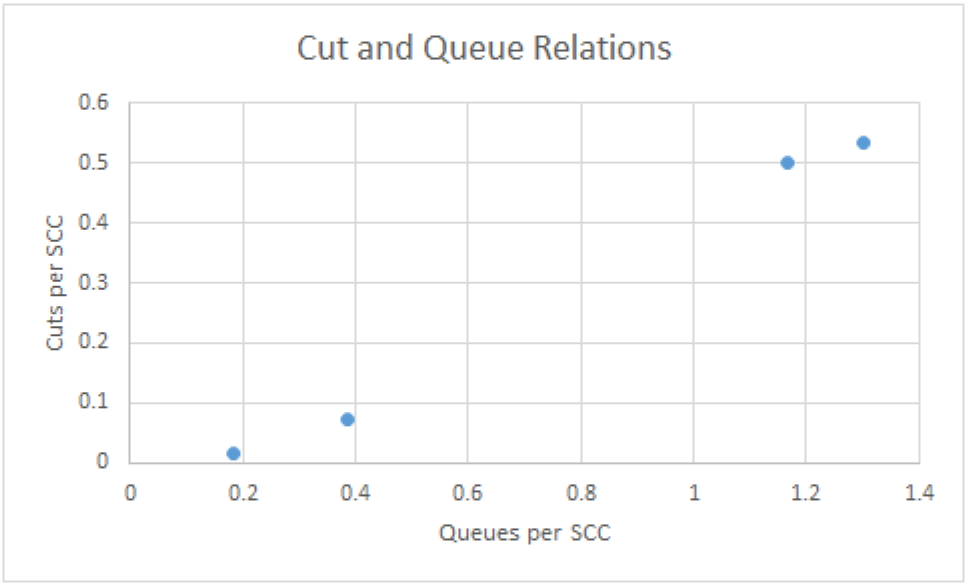


Figure 67 Overall Cut and Queues Generation Relationships

This positive linear trend in Figure 67 showed that using cut estimations for comparing algorithm performance according to queues generated, resulting in actual communication costs. By passing over 0% to 100% of a solution in hardware according to the Accumulator algorithm, it was possible to obtain the queue count for the same range of HW/SW that the Genetic, Tabu Search, and Simulated Annealing results generated.

As seen in Twill's original runs, certain percentages of hardware/software blends would generate an extremely large amount of queues, rendering the solution not lucrative at all. For example for the MIPS system, from 40% to 80% software implemented, the amount of queues would climb to the hundreds (seen in Figure 68).

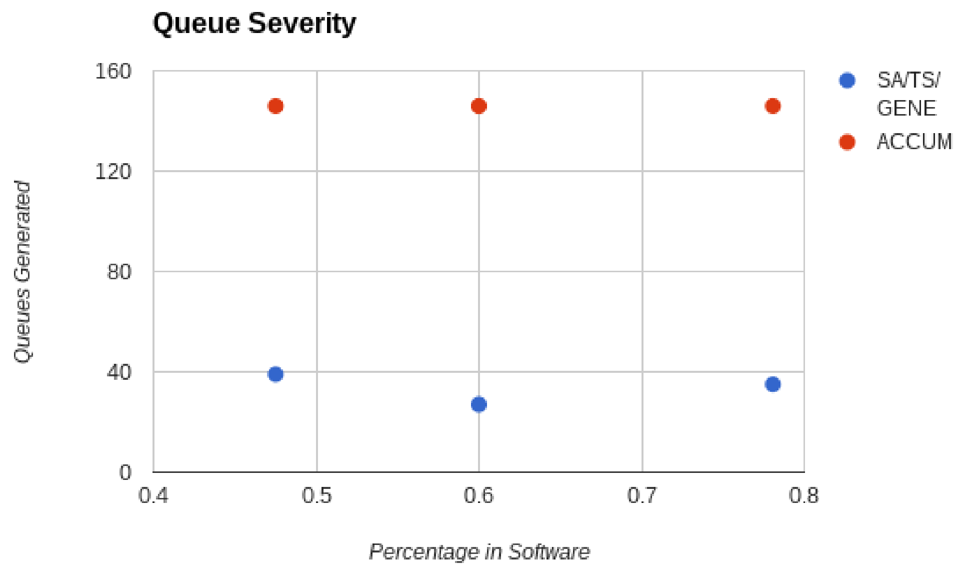


Figure 68 Queues Generated with Similar HW/SW Distributions

Combining the runs done across the same MIPS test code with Genetic, SANG, and TSSA the best case of queues generated could be found. It was on average 100 less than the original implementation. It’s clear that the ability to do multiple cuts in the PDG with solutions such as SANG, TSSA, and GA lets better solutions be explored in areas that Twill’s Accumulator algorithm would generate an extremely bad solution. This does not mean that across the board SANG, TSSA, and GA generate better solutions all the time, as Twill can still generate extremely low queue counts for other HW/SW percentages.

However as noted, previously unexplored areas of HW/SW distributions can now be explored.

8 Conclusion

Given the implementation and evaluation of the algorithms individually, it's time to compare the algorithms and observe which partitioning algorithm out of Tabu Search Simulated Annealing, Genetic Algorithm, and Twill's original accumulator, are the most effective and why. The main focus will be on each algorithm's ability to generate a low number of cuts while still delivering a heterogeneous solution. Further work that can be done on Twill with these new algorithms will be covered as well.

8.1 Tradeoffs

With solution generation in heterogeneous automation, the balance between time taken to find a solution and the solution quality is important. In the domain of time taken for each solution, we can see that TSSA is the strongest contender in time taken on the Fitness and Performance front. This is due to it's trait of "thrashing" that embraces pseudo randomness as a solution, but still manages to shape it properly using the performance algorithm.

With regards to the quality, the TSSA solution trumps in time, but on larger code samples the time taken to get to an ideal solution increases. With larger, more dynamic code bases, the time taken to reach the optimal solution will increase for all algorithms. With this knowledge we can create the following graph showing the effect of input code size with time taken to reach an optimal solution. An optimal timing solution is the solution that stays static so that out of the whole time spent running the program, ¼ of the time is spent on the same solution. At this point, further changes to the solution are an unlikely possibility.

8.2 The Cause: Cost Model or Algorithm?

Using a different cost models inside the algorithms was explored, but the solutions generated did not help GA's performance ratings or TSSA's fitness ratings. This was done to see if the cost model or algorithm was the cause of differences in the Genetic and TSSA algorithms. To do this, the comparisons in the LLVM Transform were swapped from performance to fitness and fitness to performance respectively. This means that the Genetic algorithm could be run but the performance would be used to rate the solution and decide which population members would be mated. The resulting solutions would still have both their fitness and performance ratings regardless, so these could both be extracted.

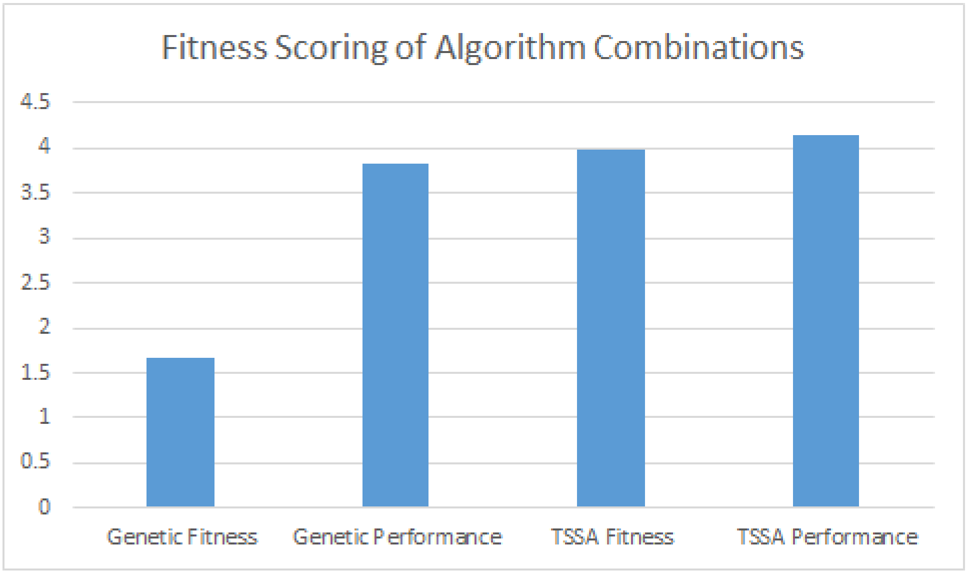


Figure 69 Combinations of Algorithms and Cost Models with Fitness Rating Outcomes

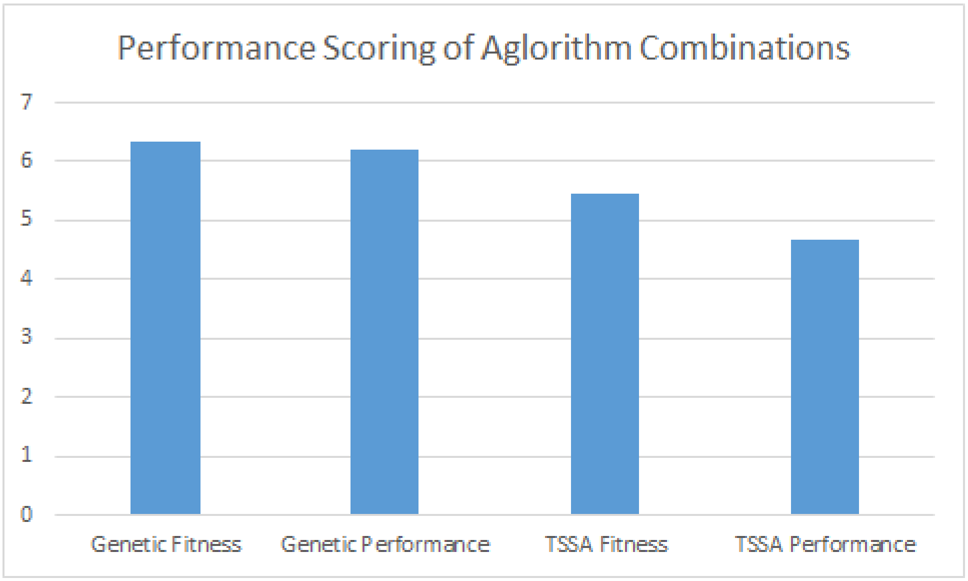


Figure 70 Combinations of Algorithms and Cost Models with Performance Rating Outcomes

As seen in Figure 72 and 71, using the vanilla Genetic Fitness algorithm resulted in the best fitness solution, while changing TSSA's comparison from performance to fitness did not modify the solution's fitness rating much. The Genetic Algorithm using the Performance rating did result in harming the fitness rating however. In the space of Performance ratings, using Fitness ratings in TSSA did not harm the results as much as Genetics' blending with Performance. The performance ratings of Genetic were left largely unchanged.

Overall swapping the cost model did tend to harm their "normal" ratings, being TSSA's performance and GA's fitness respectively. For the Genetic algorithm, the cost model swap ended up harming the results much more, increasing the rating by 160% as opposed to 15% for TSSA's swap. As such, it can be said that TSSA appears to handle different cost models better, merely using them as shaping, while genetic search does not benefit from a different cost model as much.

8.3 Summary

Examining the data shown throughout this thesis, it is clear that the optimal algorithm to use is TSSA due to its excellent tradeoffs with regards to solution time and solution quality, it's minimization of graph cuts, it's ability to examine high HW and high SW count solutions quickly, and it's ability to rapidly search a large graph using a fast local search.

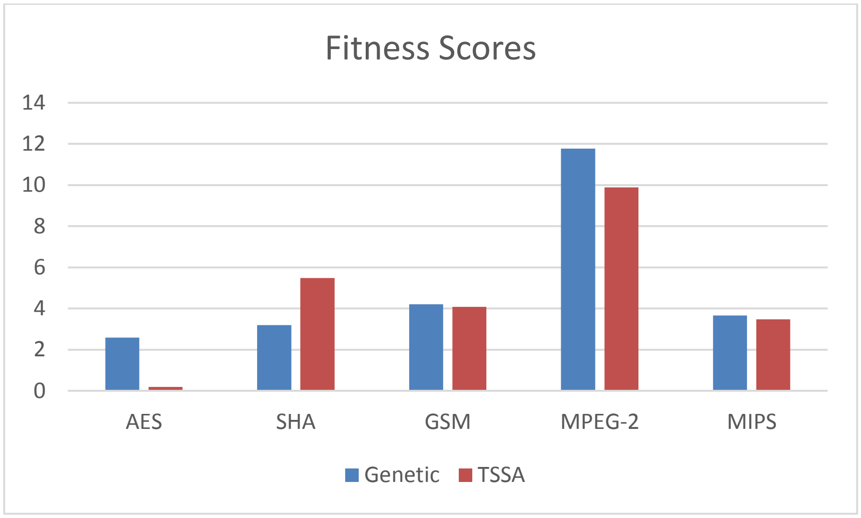


Figure 71 Fitness Scoring with CHStone Tests

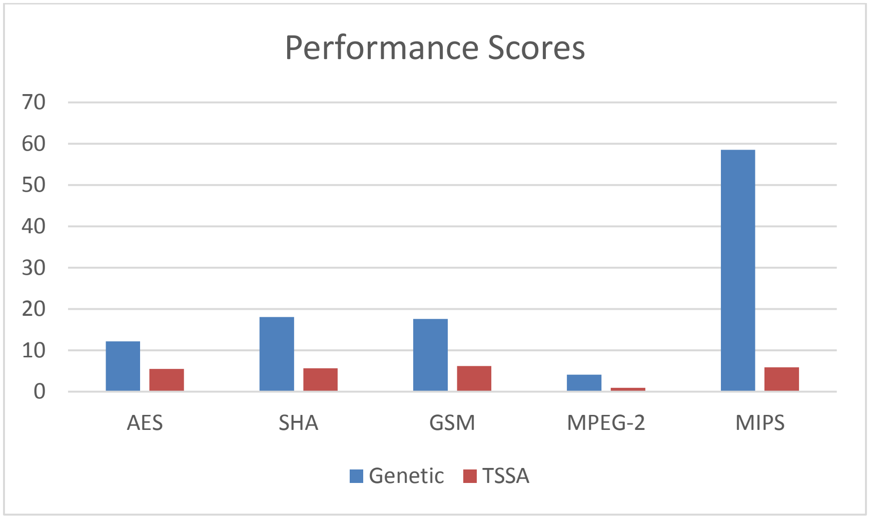


Figure 72 Performance Scores with CHStone Tests

Examining the scores of various CHStone tests in Figures 69 and 70, an outcome similar to the cost models was examined. Performance was dominated by TSSA, and the best Fitness scorings also came from TSSA as well. The TSSA algorithm was the most dependable algorithm compared to the Accumulator and Genetic Search algorithms because of its consistently strong Performance scores along with its generally stronger Fitness scores

8.4 Future Work

This thesis focused mainly on the prospects of different partitioning algorithms being used inside Twill (or other automatic heterogeneous programs) and their results. Overall this work achieved comparing and examining the options at hand, but there are still other important areas to cover in the space of heterogeneous compilation. The cost models used with the algorithms were fairly basic, and factors such as software/hardware costs and software/hardware time tradeoffs were not considered or expanded upon. Further work in this area can lead to better estimations and more accurate solutions when multiple constraints come into play. The algorithms and cost models here were fairly unconstrained. The prospects of thread scheduling was not covered either in this dissertation, and scheduling is massively important in heterogeneous computing. Combining a well-defined cost model and scheduling algorithm with the partition algorithms covered in this thesis will pave the way to a formidable automatic heterogeneous compiler. Along with these potential exploration points, the full toolchain of Twill was not used. The final component of turning the partitioned code into a bit stream to be loaded onto an FPGA and tested was not done due to basic block malformation caused by the recompilation after the partitioning algorithm finished its analysis. Implementing this final stage of Twill correctly will help solidify the data found in this investigation, and will answer the question as to whether the cost models can accurately model solution performance on Twill.

WORKS CITED

[1] D. Gallatin, "TWILL: A Hybrid Microcontroller-FPGA Framework for Parallelizing Single-Threaded C Programs," San Luis Obispo , CA: Cal Poly San Luis Obispo Digital Commons, 2014.

[2] H. Li et al., "Performance modeling in CUDA streams- A means for high-throughput data processing," in *Big Data, 2014 IEEE International Conference on,* Washington, DC, 2014.

[3] R. Hameed, "Understanding Sources of Inefficiency in General-Purpose Chips," in *ISCA '10 from Saint-Malo, France,* 2010, pp. 37-47.

[4] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," in *International Solid-State Circuits Conference*, 2014.

[5] N. Ravi, NEC Labs. America, Y. Yang, T. Bao and S. Chakradhar, "Semi-automatic restructuring of offloadable tasks for many-core accelerators," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference,* 2013, pp. 1-12.

[6] R. D. Wittig, "OneChip: An FPGA Processor With Reconfigurable Logic," in *Field-Programmable Custom Computing Machines*, 1995.

[7] J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Field-Programmable Custom Computing Machines,* 1995, pp. 12-21.

[8] M. Bilal, " ISOMER: Integrated Selection, Partitioning, and Placement Methodology for Reconfigurable Architectures," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on,* 2013, pp. 755-762.

[9] A. Putnam, "CHIMPS: A C-Level Compilation Flow for Hybrid CPU-FPGA Architectures," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on,,* 2008, pp. 173-178.

[10] N. Goulding-Hotta, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," in *Micro, IEEE Volume 31 Issue 2,,* 2011, pp. 86-95.

[11] F. Vahid, "A Decompilation Approach to Partitioning Software for Microprocessor/FPGA Platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE),* 2005.

[12] D. Andrews, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," in *Micro, IEEE Volume 24 Issue 4,,* 2004, pp. 42-54, 2004.

[13] M. Rupp, "Static Estimation of Execution Times for Hardware Accelerators in Systems-on-Chips," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on,* 2005, pp. 62-65.

[14] S. Banerjee, "Integrating Physical Constraints in HW-SW Partitioning for Architctures With Partial Dynamic Reconfiguration," in *IEEE*

Transactions on Very Large Scale Integration Systems, vol. 14, no. 11, 2006, pp. 1189-1202.

[15] D. Andrews, "Architectural Frameworks for MPP Systems on a Chip," in *International Parallel and Distributed Processing Symposium,* 2003.

[16] C. Carreras, "A Co-Design Methodology Based on Formal Specification and High-level Estimation," in *CODES '96 Proceedings of the 4th International Workshop on Hardware/Software Co-Design,*, 1996, pp.28.

[17] W. Jigang, "One-dimensional Search Algorithms for Hardware/Software Partitioning," in *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on* ,2007, pp. 149-158.

[18] H. Li, " An Adaptive Evolutionary Multi-Objective Approach Based on Simulated Annealing," in *Massachusetts Institute of Technology,* 2011.

[19] F. Ferrandi, "Mapping and Scheduling of Parallel C Applications with Ant Colony Optimization onto Heterogeneous Reconfigurable MPSoCs," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, 2010, pp. 799-804.

[20] S. Banerjee, "Efficient Search Space Exploration for HW-SW Partitioning," in *ACM CODES+ISSS 04 at Stockholm, Sweden,* 2004, pp. 122-128.

[21] H. Baranga, "Software/Hardware Partitioner," in *9th RoEduNet IEEE International Conference 2010,* 2010, pp. 252-257.

[22] A. Mishra, "Hardware Software Partitioning of Task Graph Using Genetic Algorithm," in *IEEE International Conference on Recent Advances and Innovations in Engineering,* 2014.

[23] G. Lin, "An Iterative Greedy Algorithm for Hardware/Software Partitioning," in *2013 Ninth International Conference on Natural Computation,* 2013, pp. 777-781.

[24] H. Kacem, "From UML/MARTE to RTDT: A model driven based method for scheduling analysis and HW/SW partitioning", in *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on,* 2010, pp. 1-7.

[25] H. Javan, "On The Hardware-Software Partitioning: The Classic General Model," in *IEEE CCECE/CCGEI. Ottowa,* 2006, pp. 1922-1925.

[26] J. Curreri, "Performance analysis with high-level languages for high-performance reconfigurable computing," in *Field-Programmable Custom Computing Machines,* 2008, pp. 23-30.

[27] F. Firouzi, "A Linear Programming Approach for Minimum NBTI Vector Selection," in *GLSVLSI, Lausanne, Switzerland,* 2011, pp. 253-258.

[28] K. Papadimitriou, " Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," in *ACM Trans. Reconfig. Technol. Syst.*, vol. 4, no. 4, 2011.

[29] M. Holzer, "Static Estimation of Execution Times for Hardware Accelerators in System-on-Chips," in *IEEE 0-7803-9294-9/05,* 2005.

[30] L. Wu. O. Polychroniou, "Energy Analysis of Hardware and Software Range Partitioning," in *ACM Trans. Comput. Ssyt. 32, 3, Article 8,*2004, pp. 24.

[31] J. M. S. M. Gokhale, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," in *Field-Programmable Custom Computing Machines*, 1998, pp. 126-135.

[32] M. Platzner, "ReconOS: An RTOS supporting Hard and Software Threads," in *Field-Programmable Logic and Applications*, 2007, pp. 441-446.

[33] S. Gupta, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design*, 2003, pp. 461-466.

[34] A. Canis, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor Accelerator Systems," in *ACM Trans. Embed. Comput. Syst. 13, 2, Article 24,* 2013.

[35] S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Symposium on Code Generation and Optimization*, 2004, pp. 75-88.

[36] P. Liu, "Integrated Heuristic for Hardware/Software Co-design on Reconfigurable Devices," in *2012 13th International Conference on Parallel and Distributed Computing, Application and Technologies*, 2012, pp. 370-375.

[37] Y. Hara, N. U. N. Grad. Sch. of Inf. Sci., H. Tomiyama, S. Honda and H. Takada, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on,* 18-21 May 2008, pp. 1192 - 1195.

APPENDICES

A Definitions Restated

SCC - Strongly Connected Component, a representation of an Instruction

Heterogeneous Solution - A program that is divided into hardware and software

Hardware/Software Partition - The portion of a heterogeneous solution in hardware/software

Program Dependence Graph - The graph of SCCs that is generated and used to make the software and hardware partitions

SCC Instruction Node - Also called an SCC, these contain the instruction and pointers to the next instruction.

Directed Acyclic Graph - A graph of instructions that exists initially as a software only homogenous solution

Partitioning Algorithm - An algorithm that splits up a homogenous software solution into a heterogeneous solution. Made up of a heuristic and a cost model

Solution Node - A singular heterogeneous solution intended to be used in a search graph, contains extra metadata describing the solution enclosed within

Heuristic - A function used to move across a graph of solution nodes in a manner dictated by a cost model

B Early Rating Comparison

Herein lay some early results of comparing ratings and scorings before the final results shown in the conclusion. Exploring these previously uncharted HW/SW distributions that were off limits to Twill's Accumulator algorithm, it was desired to see whether GA, SANG, or TSSA was a stronger algorithm according to their self-defined cost model. Both the Performance rating and Fitness rating were applied to all solutions generated in order to study which solution would generate the strongest across-the-board solutions.

B.1 Using the Performance Rating

In order to see which algorithms generate the strongest solution, all the algorithms were scored according to performance and fitness as ratings. These values were then compared with one another to answer the question: does using one algorithm over another get us a better overall result? The performance ratings for a given code sample were collected and normalized to avoid differences in scoring due to changes in the encountered SCC nodes. In Figures 73-78, code samples are shown with their performance and fitness ratings. Keep in mind that performance ratings are rated to have lower solutions being better as performance ratings were generated with penalties and communication costs in mind. Fitness ratings will also follow a lower is better rule, as fitness ratings are made up of the latency overhead.

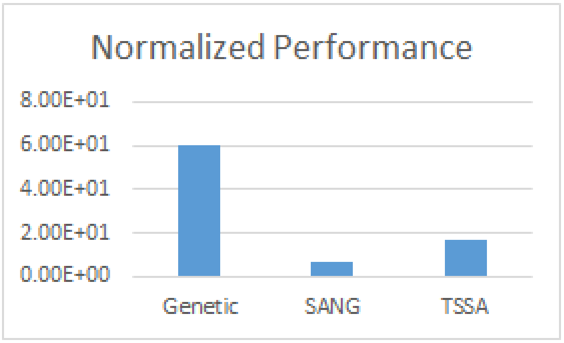


Figure 73 MIPS Performance Scores

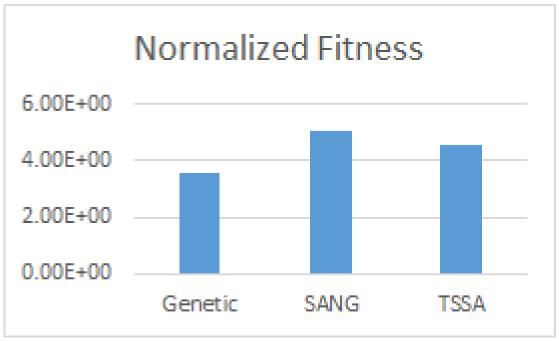


Figure 74 MIPS Fitness Scores

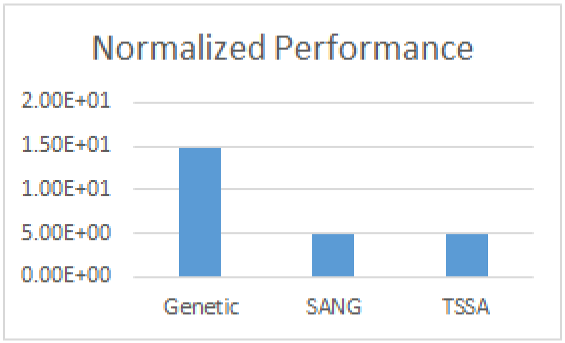


Figure 75 MPEG-2 Performance Scores

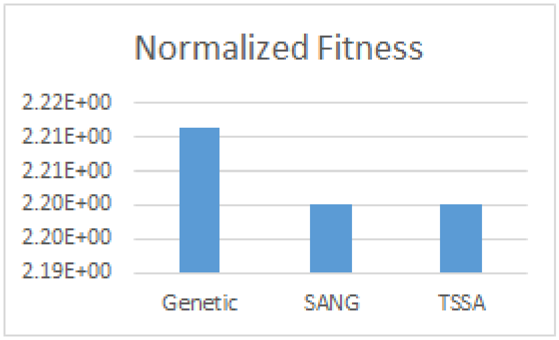


Figure 76 MPEG-2 Fitness Scores

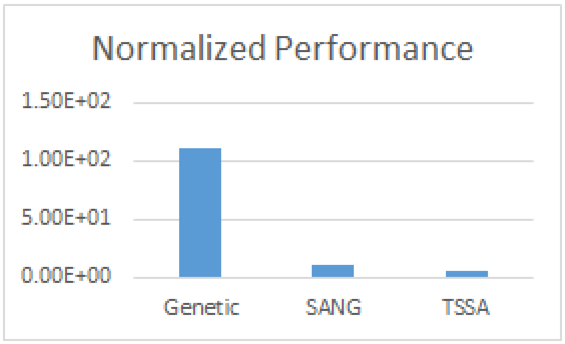


Figure 77 AES Performance Scores

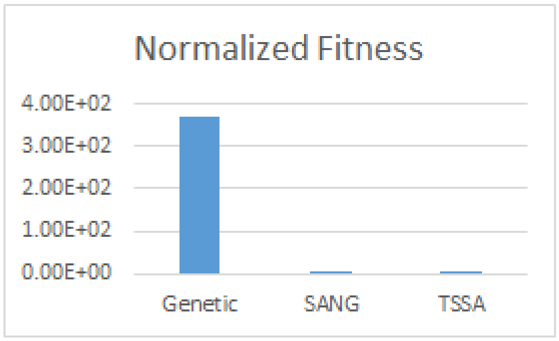


Figure 78 AES Fitness Scores

It is clear here that the genetic algorithm, while generating very good results compared to the accumulator, underperforms in even its own fitness scoring at times. This makes using TSSA or SANG quite lucrative. Granted, SANGs use of inverting SW/HW SCC instructions for a given solution's children ("thrashing") results in a quick traversal of high amounts of SW and high amounts of HW early on in it's algorithm. The genetic takes time to explore the high HW domain, as the mutations must occur and survive the high SW cost solutions first.