Optimizing Chip Multiprocessor Designs Using Genetically Programmed Response Surfaces

A Thesis
in STS 402

Presented to
The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
Of the Requirements for the Degree
Bachelor of Science in Computer Science

by

Henry Cook

STS 402
March 28th, 2007

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for papers in Science, Technology and Society Courses.

Signed_____

Approved _____ **Date** _____
STS Advisor – **Benjamin Cohen**

Approved _____ **Date** _____
Technical Advisor **- Kevin Skadron**

Table of Contents

List of Tables and Figures iii

Glossary of Terms iv

Abstract v

I. Introduction 1

II. Social and Ethical Implications 6

III. Review of Technical Literature 10

IV. Overview of Project Methodology 17

V. Results 23

VI. Conclusions and Future Work 28

Bibliography 31

Appendix A. Genetic Programming Methodology 33

Appendix B. Table of Resulting Equations 38

List of Figures and Tables

Figures

Figure 1. Overview of response surface methodology. 11

Figure 2. Mean global error with 50 sample points. 24

Figure 3. Comparison of accuracy scaling between GP and ANN. 26

Figure 4. Genetic programming tree encoding. 33

Figure 5. Genetic programming crossover operation. 34

Figure 6. Tuning parameter allocation. 35

Figure 7. Flowchart for genetic programming algorithm. 37

Tables

Table 1. Design variables used in the processor optimization study. 23

Table 2. Approximation functions and their significant variables. 38

.

Glossary of Terms

approximation function – An analytical model representing the interaction between design variables and a measure of performance. Used to predict results in place of a detailed numeric simulation. [25]

chip multiprocessor – A single chip which has been partitioned into segments, with each segment containing a separate processor. The processors may communicate via some sort of on-chip interconnection network and may share parts of the cache and memory hierarchy. [17]

core – A single processing unit in a chip multiprocessor. [17]

Design of Experiments – A family of methodologies for formulating the plan of evaluation points for an experimental study. [26]

design variable – An aspect of a design which could potentially be configured to multiple values. A simple processor example might be L1 cache size, which could be made 64 kilobytes, 128 kilobytes, or 256 kilobytes. A chip multiprocessor example might be the interconnection network topology, which could be made a torus, a mesh or a ring. [6]

design space – The set of all possible values of all possible design variables. Might be envisioned as a high dimensional space in which every design variable is a dimension and every point represents a possible design configuration. [6]

genetic algorithm – A technique based on evolutionary biology used to find solutions to global optimization or search problems. [7]

genetic programming – A technique based on evolutionary biology used to optimize a population of computer programs according to their ability to perform a computational task. In our case, the 'program' is the equation representing the response surface, and the 'task' is to match the data points obtained from full-scale simulations. [7]

Moore's Law – An empirical observation rather than a real law of nature. States that the number of transistors which can be placed on an integrated circuit will double roughly every eighteen months. Common knowledge equates transistors doubling with processing power doubling as well. [19]

response surface methodology – Explores the relationship between several explanatory variables and one or more response variables. The goal is to use experimental design to reduce the number of experiments required to construct a non-biased approximation function. [3]

response surface – The specific approximation function expressing the relationship between several explanatory variables and one or more response variables. Created from experimental data points and used to interpolate results between those points. [3]

Abstract

Putting multiple computer processors on a single chip (creating a “chip multiprocessor”) has been widely accepted as the only scalable way to further increase per chip processing power. However, the design of such chip multiprocessors is extraordinarily complicated. Computer architects must continue to consider each single processor’s design characteristics, as well as address many new design decisions once associated only with multi-chip supercomputer designs. This thesis project investigated the potential of a design optimization technique based on genetically programmed response surfaces which could significantly ease the process of improving the designs of these complicated processor architectures.

The family of techniques known as response surface methodology has been previously used in other fields to address similar design optimization problems. This thesis project demonstrates that response surfaces, specifically genetically programmed ones, can successfully be used in the computer architecture domain for rapid and robust design optimization. I found that using genetically programmed response surfaces reduces the number of full-scale simulations required to optimize a design by three orders of magnitude, while simultaneously giving the designer insight into which design variables have the most significant impact on performance. By illustrating the effectiveness of this technique and comparing it to the alternatives, I have provided a proven basis for creating tools that will soon be required for high-performance, low-energy and multi-purpose computer processor design.

Chapter I – Introduction

Throughout the past several decades, integrated circuit manufacturers have placed an ever increasing number of transistors on the chips used to create computer processors. This ceaseless and exponential increase in available on-chip resources has been fundamental to advances in computer processor design and architecture, as well as computer technology as a whole [17]. However, the transistors on a chip are just the raw building blocks – it is the task of computer architects to realize their potential through clever and effective design. As architects have had more transistors placed at their disposal, the designs they have produced have naturally grown increasingly powerful and complex. However, processor designs are rapidly becoming so complicated that automated techniques will soon be required to aid architects in making design decisions about the optimal configurations for many of the elements in their prototype designs. This thesis project presents a design optimization technique based on *genetically programmed response surfaces* to aid architects with complex processor design problems. My objective was to validate an established technique in a new problem domain by testing it on realistic computer processor design problems and comparing its performance to the few alternative methods which have been recently proposed as potential solutions.

To understand the motivation behind research projects like this one, it is essential to look at the course which computer technology has taken over the past several decades. The phenomenon of biyearly exponential increases in processing power (popularly termed *Moore's Law*) has been called both a self-fulfilling prophecy [19] and a force of technological determinism [4] due to its apparently unstoppable momentum. As computers have become increasingly powerful, available, and cheap, they have rapidly

become embedded into the very infrastructure of our society. For businesses, scientists, and many private individuals it is hard to even imagine *not* wanting a faster computer next year. Even within the industry, roadmaps and development cycles often take exponential processing growth for granted. Critically, technological progress and processor speed have become inextricably linked in the minds of both consumers and developers. Placed in such a context, with huge economic, scientific and military gains riding on the back of processor speed improvements, it is easy to understand the pressure computer architects feel. The demand for processing power continues to grow, the number of transistors on chips continues to multiply, and the only way for the architects to translate the transistors into effective processing power is to make their designs increasingly complex.

Chip multiprocessors are an up-and-coming new style of processor architecture which serves as a topical example of the ever growing complexity of computer processor designs. While they have enormous potential for performance growth, they threaten to make old design methodologies untenable. Furthermore, they have been accepted by both industry and academia as the *only* solution to the inherent inability of traditional microprocessor designs to scale with future chip manufacturing processes [17]. The design of these chip multiprocessors is complicated because architects must continue to consider not only the old single processor *design variables*, but must also simultaneously address a bevy of new design decisions once associated only with multi-chip supercomputer designs.

Further complicating matters, chip multiprocessors may be ‘homogeneous’ or ‘heterogeneous’. Homogeneous chip multiprocessors contain many copies of the same processor design on one chip (all the processing *cores* are identical). Heterogeneous chip

multiprocessors may contain a mix of many different types of processors (processing cores may be different from one another). Heterogeneous designs exponentially complicate the problem by increasing the number of potential designs that must be considered, but they have been shown to confer substantial performance boosts [8] and are therefore desirable.

This thesis project investigated the potential of a design optimization technique which could significantly ease the process of improving the designs of these complicated chip multiprocessor architectures. Processor architects could use this technique on prototype designs under development in order to optimize the designs in terms of speed, efficiency, power, heat, area, or any other characteristic which the architects care to measure and are able to simulate. Architects interested in specific domains could also use this technique to perform design optimizations with certain processor workloads in mind (e.g. genomic computational biology) or subject to certain constraints (e.g. hard power limitations). As I will discuss later, my technique is therefore useful to architects in industry and academia who are designing processors for high-performance or low-energy computing.

Optimization tools are obviously beneficial to designers, but all such tools require the collection of large volumes of performance data. Simulations that provide a detailed analysis of a potential design's performance are time consuming and computationally intensive. Running a full scale simulation of every possible design in order to find the optimal one is completely infeasible when there are billions or trillions of potential component configurations. As architects, we would rather fully simulate many fewer potential designs, and instead have some way of estimating or interpolating the

performance of the other possible designs without having to actually simulate each one of them individually.

The modeling and simulation research community has developed the family of techniques called *response surface methodology* to address exactly such situations. Given a small sample of collected performance information, these techniques build approximation functions (or *response surfaces*) which allow the scientist or engineer to estimate the performance of all the other candidate designs. This 'surface' can then be used to rapidly optimize the design using traditional optimization techniques.

Response surface methodology has previously been used in other science and engineering fields to address design optimization problems like mine where the functional computation costs (i.e. the time taken up running simulations) are high [20]. Furthermore, response surfaces created specifically by *genetic programming* have been shown to be capable of representing complex *design spaces* similar to the ones created by heterogeneous chip multiprocessor architectures [1, 25]. There is very strong potential for these techniques to be successful when applied to the pressing design problems faced by today's computer architects. Chapter 3 reviews the literature dealing with the specifics of these techniques in more detail.

The critical question I attempted to answer was whether genetically programmed response surfaces allow for rapid and robust design optimization of chip multiprocessors. This is such a new area of research that even demonstrating that these techniques are applicable within the field of computer architecture is a significant result. However, my goal was to further establish the merits of our technique by comparing it to the few alternative methods which have recently been proposed as potentially useful for chip multiprocessor design optimization [6, 11].

Programming the genetic programming of the response surface and subsequent optimization process into a design optimizing software program was also non-trivial part of this work. However, since my objectives for this thesis project included a cross-validation using data collected by another architecture optimization research project, the true deliverable of this project might be the analysis of the genetic programming-based technique as compared to the other recently proposed methods [6, 11]. The point of this comparison is to ensure that insights gained from our study will be of value to architects faced with the difficulties of heterogeneous chip multiprocessor design, even if they may ultimately choose not to use my specific optimizer software tool.

The rest of this thesis report contains a more in depth discussion of the broader social context and ethical implications of my work in Chapter 2, as well as an overview of the relevant technical literature in Chapter 3. In Chapter 4 I provide a detailed explanation of the mechanics of the genetic programming optimization technique, both in terms of the general processes of genetic programming and in terms of the specifics of the method used in this project. Chapter 5 contains the results of my experimental study and shows that my technique generates valid solutions of comparable quality to those provided by the other techniques. I conclude with a discussion of the future work for which this project has successfully laid the groundwork.

Chapter II – Social and Ethical Implications

For the sake of simplicity within this section, allow me to abstract the design process proposed by this thesis as simply an automated method for designing computer processor chips. Some relevant attributes of this method are that it can be applied to the up-and-coming generation of processor architectures, and that it can efficiently optimize designs to target specific workloads or operate under certain constraints. In this section I will explore the potential social and ethical impacts of design techniques like the one investigated by this project which lead to increased processor performance and efficiency.

Any discussion of processor performance in the past fifty years is bound inextricably to the phenomenon popularly known as “Moore’s Law.” Unlike Ohm’s Law or Newton’s Laws of Motion, Moore’s Law is not a law of nature, but rather a simple empirical observation. In a 1965 paper, Gordon E. Moore noted that the density of transistors that could be placed on an integrated circuit appeared to double approximately every two years. At the time, the integrated circuit was only six years old, and Moore was "just trying to get across the idea [that] this was a technology that had a future” [15]. However, the progress of chip production technology has continued to ceaselessly and near-exactly match Moore’s prediction to this very day! There is no physical or scientific reason explaining this phenomenon, and analysts have for years predicted its imminent failure [4], only to be proven wrong by some new process or innovation. Processing speed, computer disk memory capacity and fiberoptic cable bandwidth have all likewise increased at exponential rates (though this is not always directly related to the increasing number of transistors), which enhances the popular perception of Moore’s Law as some kind of supreme driving force behind computer technology.

Obviously, over the past fifty years Moore's Law could not help but become a tremendous force within the industry, and its implications are far reaching. It has been called both a self-fulfilling prophecy [19] and a force of technological determinism [4]. The mere fact that it is called a "Law" is indicative in and of itself. The Law's longevity has given it an aura of reliability, and it is frequently used as a "method of calculating future trends as well, setting the pace of innovation, and defining the rules and the very nature of competition" [19].

Even outside the industry, Moore's Law can be felt as the insatiable drive to have next year's processors be twice as fast as those released this year. Moore's Law has given rise to a social context which is somewhat unique: consumers not only expect to pay to keep up with exponential increases in performance – they demand it. The exponential increase in hardware computing power allows for greater complexity in operating systems and software applications, which (apparently) is what users want. In fact, software engineers continue to outstrip Moore's Law with the complexity of the programs they write, leaving consumers feeling starved for computing resources even as they upgrade their equipment. This partially explains why users are eager to buy a new processor after a year or two, and may also explain the Law's very existence in that software demands could be pulling hardware technology along behind them [19]. Whether or not it is truly a good idea to give consumers what they say they want is a tough social question; I only wish to note that this thesis project is an artifact for more effectively satisfying said user demand. It is also worth noting that Moore's law in its current incarnation has been argued to be a technologically deterministic force [4] based on the observation that software applications may be considered to be good or bad,

hardware designs may be good or bad, but having more transistors and raw processing power is always unquestionably 'good'.

There are some significant economic implications of Moore's Law. In general, if a certain computer processor costs $X today, a processor of equal power will cost $X/2 next year, and be essentially worthless after five or so years [15]. This means that better computers constantly become more affordable and available to more people, but also that they are rapidly made obsolete. The availability of steadily improving products at rapidly falling prices is actually very appealing to consumers [15]. Furthermore, economists have argued that the economic boom of the 1990s is to some degree a reflection of Moore's Law. These arguments are usually related to the measured increase in productivity during this time period correlated with increased personal computing power:

> "Spread throughout the economy, higher productivity means higher wages, higher profits, lower prices. Productivity increases aren't necessarily painless... But history shows that workers displaced by productivity-enhancing technology usually find other, better jobs. In the long run raising productivity is essential to increasing the national standard of living." [15]

From an economist's perspective, it is essential for our nation's well being to continue to increase productivity, on which computer performance has been shown to have a possibly significant effect. From the hardware and software corporations' perspectives, it is essential for the bottom line to drive users to constantly desire new computers and the processors inside them. Techniques like mine, which essentially serve to 'convert' transistors into processing power, will be integral to both of these efforts.

As a final note, it is worth considering that regardless of the state of the general user, there are a huge variety of high-performance or low-energy computing tasks which will tremendously benefit from the capabilities offered by design techniques like the one I have proposed. High performance computing is often scientific or medical in nature and driven by such lofty goals as finding cures for diseases like Alzheimer's and cancer,

mapping genomes, or studying astrophysics and cosmology. Low energy computing is focused on optimizing processor designs so as to consume less power and energy. Such processors are more environmentally aware, and can also be used in affordable computing initiatives (such as One Laptop Per Child). The beneficial environmental, economic, and social impacts of such research efforts are indubitable, and improved processor designs created by my technique (or a similar one) might be essential to their long term success. As the next chapter will discuss, my proposed design technique enables optimization to certain targets (e.g. scientific workloads) or subject to certain constraints (e.g. power efficiency) and so should prove to be a useful tool for research efforts of both varieties.

Chapter III – Review of Technical Literature

The goal of this thesis project was to explore the potential effectiveness of applying a design process taken from the modeling and simulation community to a problem which has recently been encountered by the computer architecture community. For this reason, providing a proper context for the project requires an examination of two somewhat disparate threads of research from two different research areas.

The first goal of this literature review is to illustrate some of the ways that the modeling research community's approach (response surface methodology) has been used successfully in a variety of engineering domains. The specific focus here is on a response surface methodology based on genetically programmed approximation functions, because that is the technique used in this research project. The second goal of this chapter will be to enumerate the solutions which ongoing computer architecture research efforts have attempted to apply to their design problem (i.e. the computational difficulty of optimizing chip-multiprocessor designs). By illuminating the successes of the technique in other contexts, and by clarifying the shortcomings of current solutions to the architecture simulation problem, I hope to more fully the demonstrate merit of this research project.

Uses of Response Surface Methodologies

There are many cases in the science and engineering domains where a researcher would like to optimize a design's performance subject to certain constraints. An aerospace engineer might be interested in optimizing lift subject to weight overhead, a mechanical engineer might be interested in power subject to manufacturing costs, and a computer architect might be interested in performance gains subject to power

consumption. Usually in these situations various design alternatives can be evaluated by comparing the results of highly detailed numerical simulations. However, sometimes even just running simulations incurs such a high computational cost that it becomes impractical to empirically evaluate all or even a significant fraction of the many possible design alternatives. As the discussion below shows, this is precisely the case for computer architecture simulations.

The introduction of approximation concepts in the 1970s provided an alternative approach to highly detailed numerical simulation by replacing the detailed objective and constraint functions of the simulations with simplified analytical models [18]. In modern research, there are several commonly used global approximation techniques which automatically generate easily optimizable analytical models. Two examples of such

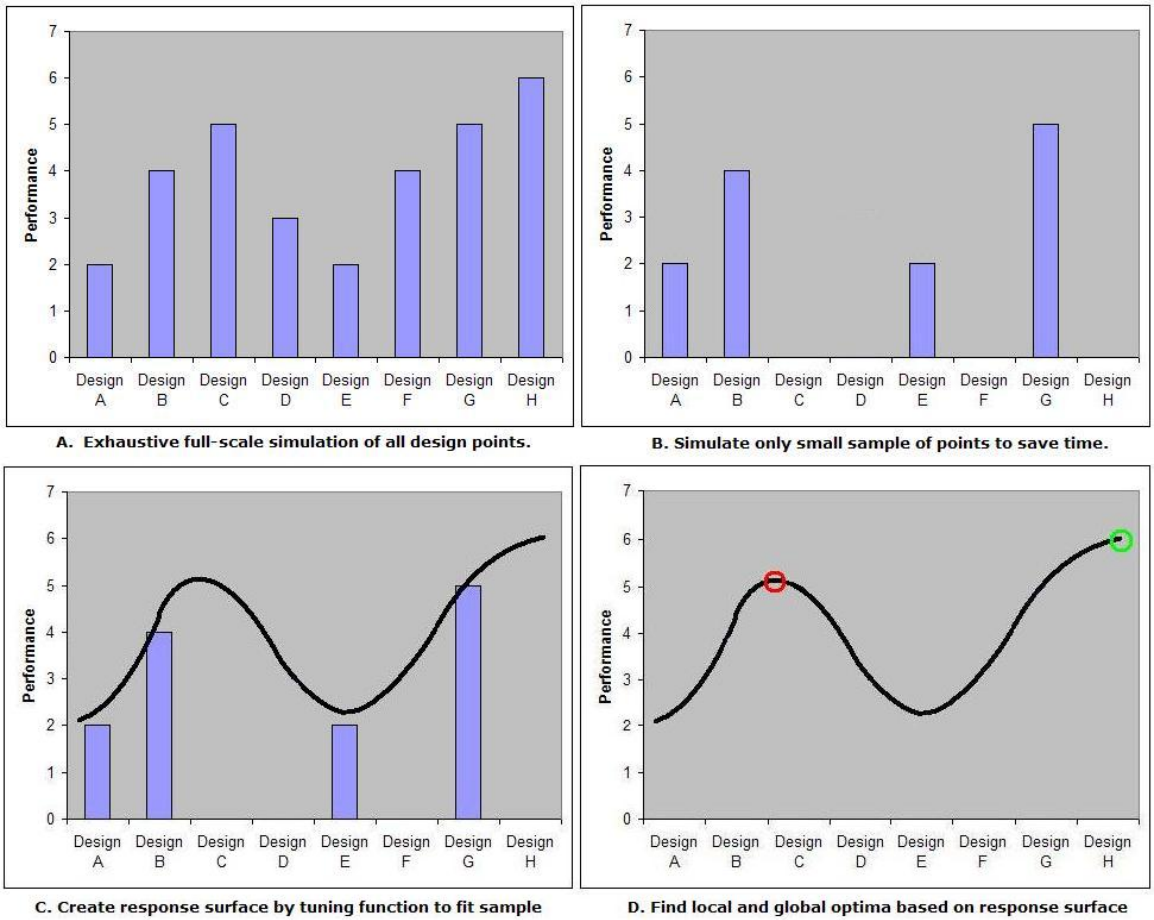

Figure 1. An overview of response surface methodology (drawn by author).

techniques are response surface methodology (RSM) and artificial neural networks (ANN) [1]. The weak points of the ANN approach will be discussed later in this section, but for now I will focus on the proven applicability of RSM to high cost simulation problems.

In general, RSM involves the creation of a polynomial approximation function that is fit via regression analysis to a set of data points collected from the *design space*. Figure 1 gives a graphical outline of this process. The goal is to create a function which allows for accurate estimation of the performance of design points which were not actually simulated. In addition to requiring many fewer full-scale simulations overall, the response surface function speeds up design optimization efforts by allowing the use of derivative-based optimization techniques, which converge rapidly [22]. However, one weakness of traditional RSM is that the structure of the approximation function (e.g. linear, polynomial, etc.) must be chosen ahead of time by the user, which lowers the quality of the approximation. Another weakness is that the selection of design variables for inclusion in the regression analysis is a combinatorial problem, and therefore computationally prohibitive [3].

A solution proposed by L. Alvarez [1] to both of these problems is to use an algorithm based on *genetic programming* [5] to determine the structure of the approximation function. Toporov and Alvarez have successfully applied this technique to the three-bar truss problem [23], Rosenbrock's function [22], the recognition of damage in steel structures [24], the approximation of design charts, a prediction of the shear strength of reinforced concrete deep beams, and an optimization of the calcination of Roman cement [1]. Their work found that genetic programming is able to provide a solution even if the implicit objective function is highly non-linear and even if a large

range of optimization variables are adopted (both of these cases are true with architecture design optimization problems). Furthermore, the genetic programming process can detect which design variables are important during the construction of the approximation model, and in certain cases can successfully extrapolate beyond the initially defined range of the variables. While genetic programming works best when designer knowledge is used to guide the process, it is nevertheless an extremely powerful automatic tool for modeling and optimizing industrial processes like those mentioned above [1], and my goal was to prove that the technique is equally applicable to the computer architecture domain.

Previous Solutions in the Computer Architecture Literature

Computer architects have long recognized that the event-driven, trace-based architecture simulations they use are extremely computationally intensive. However, their best efforts to speed up simulations in order to rapidly explore a design space (e.g. [12]) pale in comparison to the vast multi-dimensionality of the new heterogeneous chip-multiprocessor design space [23]. In recognition of this problem, several extremely recent (October 2006) research projects have addressed the task of creating global approximation functions of processor performance and power. The technique proposed by this thesis can be considered a third method for accomplishing the same goal, but one which I hope to prove is more informative, more efficient, and no less accurate.

Ipek et al. [6] propose using artificial neural networks (ANNs) to automatically create predictive global approximation models of several computer architecture design spaces. They randomly sample points throughout the design space and use the results to train their neural networks. The resulting trained networks are used to predict

performance over the rest of the design space. Results reported in their paper show that they can achieve 98-99% prediction accuracy after only training on 1-2% of the design space. Significantly, they conducted an exhaustive search of their design spaces in order to determine the true best designs, and kindly made these full-scale simulation results available to me for this research project. This allows me to directly compare my RSM technique to their ANN technique in terms of their performance on realistic simulated processor data.

In general, neural networks are a useful tool because of their ability to learn non-linear functions. However, they also represent a "blackbox" in that they give the designer no explicit insight into the relationships that the network develops between the input variables. Neural networks are also infamously bad at extrapolation (as compared to interpolation), and are susceptible to curve over- and under-fitting. Recall from above that genetically programmed response surfaces, by contrast, are both non-linear yet explicitly defined, and also have some limited extrapolation capabilities. The design space Ipek et al. search is actually relatively small, so a critical questions will be whether their technique scales well to the kind of extremely high dimensional spaces that might be expected from heterogeneous chip multiprocessor optimization problems.

Lee and Brooks [11] perform a more traditional style of regression modeling in order to generate approximation functions for performance and power. Their design search space is larger than Ipek et al.'s, especially because it includes applications as parameters, and consists of over 20 billion design points. Surprisingly, they are still assuming a relatively simplistic memory hierarchy and ignoring all interconnect design concerns, which means that including the kind of heterogeneous design decisions that will soon be required could result in trillions of possible design configurations. Their

regression curves are created from cubic splines (piecewise polynomial functions) and therefore non-linear, but may not deal well with discontinuities in the data. Nevertheless, they still achieve predictions of 96% accuracy while sampling only one in 5 million design points [11]. In contrast to Ipek et al.'s method and our proposed technique, their method is more of a statistical process than an algorithm. It is therefore not fully automated and requires some *a priori* intuition on the part of the designer [6]. Finally, their method for selecting data points for full simulation is not based on any design of experiments (i.e. this work) or detected variance (i.e. [6]), but is instead only uniform random sampling.

A third recent paper explicitly deals with the heterogeneous chip-multiprocessor design space. Kumar et al. [9] encounter over 2 billion configuration options even before factoring in permutations of applications. They had enough computing resources at their disposal to attempt a brute force global search, and contrast these exhaustive results with those obtained by a simplistic hill-climbing search algorithm. Hill-climbing search selects a configuration 4.5% worse than the configuration selected by the exhaustive global search, while requiring 86% less full simulations [9]. Kumar et al.'s methodology is fundamentally different from the previously discussed techniques and our proposed technique because they are not interested in creating any kind of approximation function, but instead search through the design space directly. While it might be interesting to attempt a direct search with a more advanced search algorithm instead of hill-climbing (e.g. simulated annealing or genetic search), in my opinion it is dubious whether any direct search algorithms can successfully scale to the heterogeneous design space size without requiring a tremendous number of full-detail simulations.

One of the main benefits of my technique is that the response surface function it creates is generated automatically, but is still explicitly stated for the architect to see. This is in contrast to the spline regression technique which does not automatically search for new approximation function formations, and the ANN approach which is automatic but which cannot provide any explicit function to the architect. The former relies too heavily on the researcher's direct intervention and prior knowledge, while the other does not provide the researcher with any insight into why any given designs were selected as optimal. In addition to a proven robustness of functionality, genetically programmed response surfaces are ideal in terms of the mix of automatic, evolutionary search and explicit user feedback which they provide. I am confident that these characteristics lead to a more robust and insightful design process than could be achieved with the other techniques.

Clearly, while this area is on the leading edge of computer architecture research, the first cut approaches have already achieved significant results. The papers discussed in this section simultaneously demonstrated the effectiveness of global optimization techniques as time-saving tools while also motivating the need for studies, like mine, which are intended to address larger design spaces and provide more efficient, explicit and robust approximation methodologies. As discussed earlier, genetically programmed response surfaces have the potential to address many of the issues raised by the current architecture optimization research projects. By illustrating the strengths of the genetically programmed response surfaces in other contexts, and by identifying the shortcomings of current solutions to the problem, I hope to have provided a strong basis for understanding the context and importance of the research results contained in the following chapters.

Chapter IV – Overview of Project Methodology

The goal of this chapter is to provide an overview of the specific methods involved in the process of using genetic programming to create robust response surfaces. Response surfaces are created from a small subset of the possible design points in order to serve as accurate approximation functions for performance across entire design space. In other words, the response surface is used by the designer to approximate the behavior of all possible designs by interpolating between a few experimentally determined values. This overview will address all the steps involved in applying the genetic programming technique to previously collected processor architecture simulation data, the nature of the underlying processes which make the technique effective at creating accurate approximation functions, and some of the experimental parameters used in my study. Because the specifics of the genetic programming subprocess itself are lengthy and involved, I discuss them in more detail in Appendix A.

Overview of Procedure Steps

Because the shape of the response surface generated by the genetic programming process is determined mainly by the few sample points that are actual simulated in full detail, it is imperative that we select sample points that are representative of the entire space. Failing to do this will inadvertently create response surface functions that leave out key characteristics of the overall pattern of behavior, leading to less accurate result predictions on a global scale.

I chose to use a *design of experiments* (DOE) technique known as the Audze-Eglais Uniform Latin Hypercube design of experiments to select the design points included in the sample set. Audze-Eglais DOE selects sample points which are as evenly

distributed as possible through the design space by formulating the problem as one of gravitational attraction and minimizing the "potential energy" of the system of points. I formulated my Audze-Eglais DOEs using the optimization technique described ny bates et al. [2]. A major benefit of this particular DOE was that I was able to specify in advance the number of points I wanted to sample, while maintaining a high confidence that they were evenly distributed across the space of all possible designs.

Normally, upon choosing their sample design points a user of my technique would take the list of points and simulate each one of them in high detail. In the case of this thesis project, I was able to use simulated performance data previously collected by [6] for their neural network-based study. Using this data saved time, and more importantly it allowed me to directly compare our techniques. The fact that the dataset I used is actually an exhaustive global search of the entire design space was also critical to verifying that my technique truly works effectively; more discussion of this comparison and validation is contained in the next chapter. Unlike the hill climbing search performed in [9], the time my technique takes to determine an optimum is in no way based on the length of time it take to run a full-scale simulation, as the simulations of the design points in the sample set can all be done as a preprocessing step.

The resulting full-scale simulation data corresponding to the selected sample subset is then used by the genetic programming algorithm to construct the response surface, which is a non-linear approximation function of the data. The specific details of the genetic programming process are laid out in detail in Appendix A and [1], but a high level summary follows:

Candidate approximation functions are represented as in-order expression trees, where an individual node in a tree represents an operator, design variable value or tuning

parameter. Operators can be defined by the user – in my case I have defined them to include simple arithmetic functions, a square root operator and a logarithmic operator. A group of randomly generated candidates forms the first 'generation' of the genetic programming process. Each candidate is then assigned tuning parameters, the tuning parameters are fitted to the data via a gradient descent technique [14], and each candidate is evaluated based on how well it fits the simulation result data for all the points of the sample subset. It is important to note that at this stage the technique fits equations to data rather than evaluating the fitness of individual processor designs. Candidate function trees with low fitness are discarded, while candidates with high fitness swap subtrees to create a new generation of candidates. Over time, the average fitness of the population increases, but mutation of the candidates allows for a relatively global search of possible approximation functions. This breeding and mutation process is repeated until the fitness of the best candidates ceases to improve or some kind of time threshold is exceeded.

The result of all this work is a set of approximation functions which are a good fit for the sampled data, and which hopefully do a good job of approximating the global design space as well. As I will discuss in the next chapter, the genetic programming technique works quite well for this purpose.

Nature of the Underlying Processes

There are several concepts which are innately associated with the methods used in this thesis project which are critical to the overall success of the technique as a whole. I would like to take the time to review some of them, because understanding their guiding effect on the search for a good response surface is essential to understanding why I selected certain methods for use in my processor design optimizer.

The concept of fairly selecting a subset of experimental points to use as a sample of global behavior is a well studied process [2]. However, previous computer architecture performance approximation research has often simply opted to choose points from the design space uniformly at random [11], or chose them dynamically at run-time based on estimates of training inaccuracies [6]. The former method risks the possibility of poor point distribution leading to an erroneous approximation function. The problem with the later method is that it is impossible to know in advance how many or which points must be sampled. The training process itself may be very efficient, but the overhead of conducting additional performance simulations in the midst of training is very high. By contrast, my methodology in this project makes use of a well studied design of experiments technique which creates a list of evenly distributed sample points based only on the design space dimensionality.

At the heart of all genetic algorithms are the biological concepts of natural selection and survival of the fittest. Knowledge about the problem domain is encoded by the fitness measure and the representation scheme of individuals' genomes, and there is a provable mathematical relationship which relates the former to the later [1]. For any fitness measure, certain subsets of an individual's genome will correspond to a better overall fitness, and certain subsets will correspond to a worse overall fitness. By allowing individuals to breed *in proportion to their fitness* (as compared to the rest of the population), the good subsets will receive exponentially increasing numbers of opportunities to be reproduced. Over time, this provably results in increasingly fit individuals which converge on a small subset of very fit solutions [1]. Simultaneously, the mutation functionality serves to prevent early convergence on a non-optimal subset by injecting random variance into the population. The push and pull interaction between

fitness-based breeding convergence versus mutation-based variance is what makes genetic algorithms so effective and efficient at locating global optima.

A critical step in the genetic programming process is to assign a set of tuning parameters to a candidate expression tree which represents the structure of a potential approximation function (see Appendix A). Using tuning parameters abstracts the structure of the expression from specific data values and results in simpler functions [1]. Tuning parameters are added to an expression tree deterministically based on the tree's topology, but the actual values of the parameters must then be tuned to match the sample data as closely as possible. For this tuning process I used an open source implementation of the Levenberg-Marquardt (LM) algorithm for nonlinear least-squares problems. LM is an iterative technique that finds a local minimum of a function that is expressed as the sum of squares of nonlinear functions. It can be thought of as a combination of steepest descent and the Gauss-Newton method [14], with the method of convergence changing depending on how close it is to a solution. The effectiveness of the overall genetic – programming technique hinges on the ability to detect fit candidates, and a poor tuner will result in theoretically fit trees being discarded inadvertently. Therefore, a good tuning algorithm is essential for good overall performance. As can be seen in the results reported in the next chapter, the combination of genetic search and LM minimization used in my thesis project seems to work well, especially for small sample set sizes.

Experimental Parameters

It is worth taking the time to discuss the experimental design parameters I used for my tests of my own genetic programming-based optimizer. For further clarification of the effect some of these parameters might have on the performance of the system,

please see Appendix A. Unless otherwise noted, my primary motivation in using the parameter values that I did was to match the values used by Alvarez in his genetic programming optimization studies [1].

I used a population size of 500 candidate trees for every generation of the genetic programming process. This was large enough to hopefully generate some interesting diversity, but not so large as to be overly inefficient. The maximum candidate tree depth was initially limited to 15 levels upon tree creation, and thereafter became limited by the number of tuning parameters the Levenberg-Marquardt algorithm could successfully optimize. The genetic search to get an initial guess at tuning parameter values used a maximum of 30 generations over 70 individuals (these values are quite small, but only an initial guess was required for the LM algorithm to do the rest of the work). The maximum number of generations allowed for the genetic programming process was 1000, but the GP always converged after fewer than 100 generations. I used a 10% elite percentage and 10% kill percentage, meaning that the best 10% of the previous generation was preserved unchanged in the next generation, and the best 90% of the previous generation was allowed to breed to populate the next generation.

These experimental parameters are the ones that I used for the tests whose results are reported in the following chapter. Future research may suggest more effective values. In general, my intuition is that there is a tradeoff between the efficiency of the algorithm and its ability to capture complicated details of the design space.

Chapter V – Results

While previous research has already shown that genetically programmed response surfaces work well to predict the behavior of complicated, non-linear, multi-dimensional design spaces, I sought to prove the effectiveness of this technique on an actual computer architecture design optimization problem. I was lucky enough to have the opportunity to get the set of exhaustive processor simulation data that was used in Ipek et al.'s ANN-based optimization study [6]. This allowed me to test my technique on a realistic design problem and perform a direct comparison to Ipek et al.'s results.

To validate their technique, Ipek et al. conducted performance sensitivity studies on memory system and micro-processor designs. They assumed a 90nm technology and modeled contention and latency at all levels [6]. The design variables for the superscalar processor study and their associated possible values can be found in Table 1.

Variable Parameters	Values
Fetch/Issue/Commit Width	4,6,8 Instructions
Frequency	2,4 GHz (affects Cache/DRAM/ Branch Misprediction Latencies)
Max Branches	8,32
Branch Predictor	1K,2K,4K Entries (21264)
Branch Target Buffer	1K,2K Sets (2-Way)
ALUs/FPUs	2/1,4/2,3/1,6/3,4/2,8/4 (2 choices per Issue Width)
ROB Size	96,128,160
Register File	64,80,96,112 (2 choices per ROB Size)
Ld/St Queue	16/16,24/24,32/32
L1 ICache	8,32KB
L1 DCache	8,32KB
L2 Cache	256,1024KB
Fixed Parameters	**Value**
L1 DCache Associativity	1,2 Way (depends on L1 DCache Size)
L1 DCache Block Size	32B
L1 DCache Write Policy	WB
L1 ICache Associativity	1,2 Way (depends on L1 ICache Size)
L1 ICache Block Size	32B
L2 Cache Associativity	4,8 Way (depends on L2 Cache Size)
L2 Cache Block Size	64B
L2 Cache Write Policy	WB
Replacement Policies	LRU
L2 Bus	32B/Core Frequency
FSB	64bits/800 MHz
SDRAM	100ns

Table 1. Design variables used in processor optimization (table from Ipek et al. [6]).

The various allowable permutations of these variables results in slightly over 20000 possible design points. Such a design space is small enough that an exhaustive search is possible for validation purposes, but large enough to be realistically complex.

Testing the performance of the candidate designs via functional simulation requires a set of reference applications. Ipek et al. use a subset of the SPEC CPU 2000 [21] applications selected to match the clusters which were identified by Phansalkar et al. [18] using principal component analysis grouping. For the memory system and processor studies, they chose a code (bzip2, crafty, gcc, mcf, vortex, twolf, art, mgrid, applu, mesa, equake, and swim) to cover each one of the Phansalkar et al. clusters, ensuring that they model the diverse program behaviors represented by the suite [6]. The performance measure I used in this study was instructions per cycle (IPC), but the other performance statistics could have been used just as well.

In order to test the global accuracy of the response surface function that the genetic programming process created, I calculated the mean and standard deviation of error between the actual simulated IPC and the IPC predicted by the response surface approximation function. These measures are similar to results reported in [6] and allow us to grasp how well the function matches the global data and the degree to which the amount of error is uniform across the entire design space. As can be seen in Figure 2, the errors reported with a sample size of only 50 design points (out of over 20000) are all on average less than 5%. This means that the approximation function generated by genetic

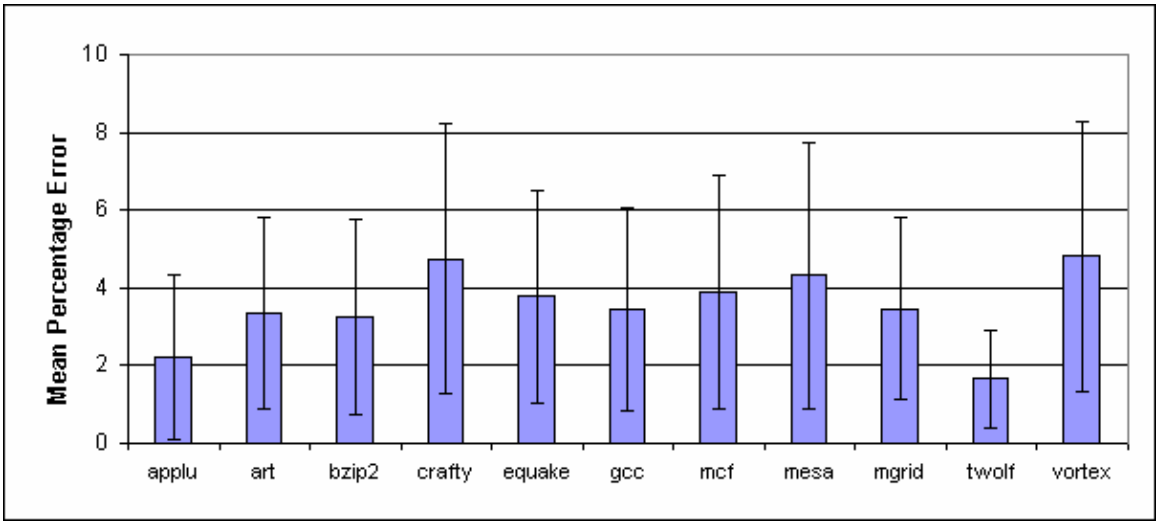

Figure 2. Mean and standard deviation in global error with 50 sample points (drawn by author).

programming is generally over 95% accurate after seeing only 0.25% of the design space. In other words, we can reduce the number of full-scale simulations by almost three orders of magnitude while still maintaining a high degree of accuracy. As Figure 2 makes clear, the genetic programming technique performs similarly well across the various SPEC benchmarks included in the study. The accuracy of the functions created from only 50 fully simulated sample points are similar in quality to those reported by Ipek et al. [6] for their ANN-based approach.

Ipek et al. made a further study of accuracy by examining how their results change as they increase the number of sample points included in their training process. I performed a similar comparison for several of the benchmarks, and, as can be seen in

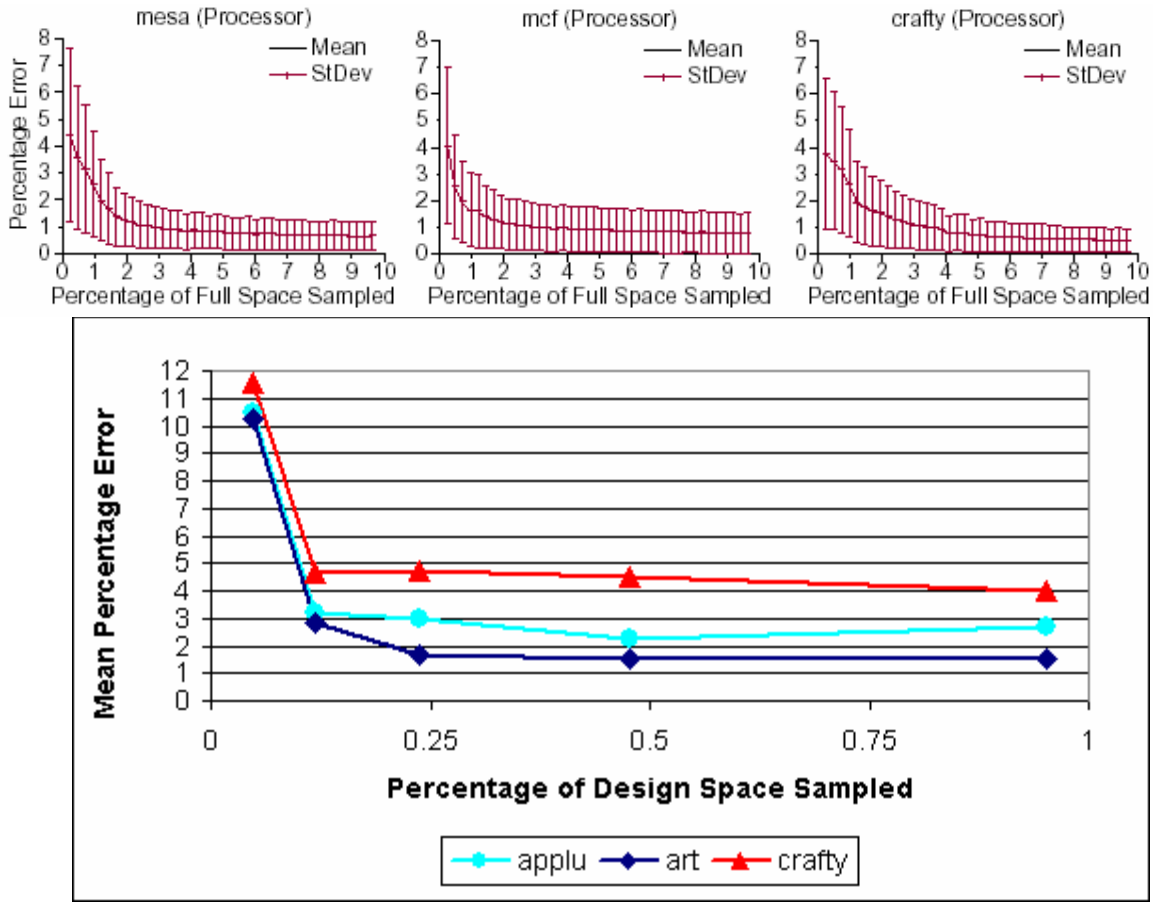

Figure 3. Comparison of accuracy scaling of GP (bottom) with ANN approach from [6] (top).

Figure 3, I found that the genetic programming technique does see improved results up to a point when more simulated data points are included. My results are generally better than Ipek et al.'s when only 0.12% or 0.25% of the design space is sampled, and my technique's accuracy remains steady with increasing sample set size. The point of sharply diminishing returns occurs at a much lower sampling percentage than for Ipek et al.'s technique – this is a good thing because it means architects will have to sample fewer points to achieve quality results. While Ipek et al. achieve slightly better accuracy on most applications, they must sample at least 2% of the design space to do so.

One of the major stated benefits of genetically programmed response surfaces is that they give the architect back an explicit function on which to perform the optimization. Having this function gives the architect insight into the tradeoffs of the design that are relevant when predicting design performance. Therefore, the architect will have a better idea of which design variables are good candidates for further study, due to their significant impact on performance. My results begin to validate this idea. According to Huh et al., *mesa* and *equake* have performance which is processor-bound (high IPC, small working set), while *art* and *mcf* have performance which is bandwidth-bound (huge working set). Looking at the response surface functions in Table 2 (Appendix B), we can see these traits reflected in which design variables are included in the actual approximation function. With the former two benchmarks, the genetic programming process identified fetch/issue/commit width as significant, while with the later two benchmarks L2 cache size was declared significant. Further testing of this theory might involve using benchmarks from the same clusters as identified by Phansalkar et al. [18] to see whether they produce approximation functions which include the same design variables.

The initial results of these tests of the genetically programmed response surface methodology for processor architecture performance modeling are very promising. The technique's performance on a realistic data set is comparable to other recently published results. While more tests need to be done for a validation of the technique to be considered complete, the technique has already satisfactorily fulfilled all the requirements which I have claimed that it would. There may be some room for improvement, but at this juncture it is certainly fair to conclude that the technique is a successful method for approximating performance for the purposes of architectural design optimization.

Chapter VI – Conclusions and Future Work

In this thesis project, I have applied an optimization technique based on genetically programmed response surfaces to the complicated problem of processor architecture design optimization. This technique attempts to develop a robust response surface based on data from only a small number of full-scale simulations that accurately predicts the performance of all possible design configurations. Such an approximation function reduces the number of costly simulations required to optimize a design, aids in the customization of a design to match a target application workload, and provides the computer architect with insights into the nature of performance tradeoffs within the design space.

The genetic programming methodology successfully creates very accurate models of the behavior of IPC performance in a realistic single superscalar processor design space using a sample which is only a tiny percentage of the overall size. These results are comparable to those obtained by Ipek et al. [6] with a similar ANN-based approach. My technique appears to be up to twice as accurate as theirs at small sample sizes, but is just as accurate, or slightly worse, than theirs when larger samples are available. However, my technique has the added benefit that it provides the designer with the explicit function being used to approximate the design space. This may give the computer architect further insight into which design variables are good candidates for optimization or improvement, due to their significant impact on performance.

A fundamental difference between my technique and a direct search through the design space (i.e. with hill-climbing [9] or a normal genetic algorithm search) is that with this technique the user has to do only a small number of sample simulations in advance, and then never has to do any more. With direct searches, each new generation or

movement will require another set of full-scale simulations to determine the fitness of all the new configurations. In my opinion, such methods will not scale effectively to the huge multi-dimensionality of the heterogeneous chip multiprocessor design space in which so many architects are now becoming interested. In contrast, response surface methodology provides a solution because it requires only a small number of simulations, while the genetic programming component ensures satisfactory levels of accuracy.

In addition to a proven robustness of functionality, genetically programmed response surfaces seem to be ideal in terms of the mix of automatic, evolutionary search and explicit user feedback which they provide. I am confident that these characteristics lead to a more robust and insightful optimization process than could be achieved with the other techniques. Because of their explicit approximation function and ability to identify design variables significant to individual applications' performance, I feel that this technique will be of great interest to computer architects interested in low-power or high-performance chip multiprocessor design.

At this time, the technique has been tested on a realistic but somewhat small architecture design problem. The results of these initial tests are very encouraging and indicate that the technique does in fact have as much potential as I originally proposed. While the technique currently fails to improve the quality of its approximations when given more sample data points, I am certain that future refinements will address this problem and improve the already high accuracy of the approximation provided by the response surface. In general, the technique has performed as well as I could have hoped on its first architecture design optimization problem, and I am eager to apply it to harder problems in the near future.

Some immediate future work on this project will be to attempt to improve the accuracy of the approximation functions when they are given additional sample points. This might be done by adjusting the parameters of the genetic programming process to discourage rapid convergence, or possibly by including alternative methods for adjusting the tuning parameters. More detailed sensitivity studies should help refine the process as presented in this thesis. Other future work might address larger design spaces or optimizations of multiple performance measures at once. An as yet unfounded intuition of mine is that the most efficient way of doing this kind of architecture design optimization might be to use my genetic programming technique to get a response surface based on a very small initial sample, use the resulting response surface to find some interesting global minima, and then use a genetic or hill-climbing search locally around those minima to find the actual optimal configurations.

Techniques like the one I have presented in this thesis project will be critical to address the tough processor design optimization problems currently facing the industry as they struggle to keep up with Moore's Law. Societal demand for increased computational power is continuing to rise, and computer architectures are by necessity growing ever more complicated, so it seems inevitable that computer architects will find effective design optimization tools more essential to their work then ever before. Happily, the first generation of techniques for automated processor design optimization, including the one presented in this project, have already produced significant results. In the future, I believe that we can look forward to refinements and hybrids of these techniques producing truly unprecedented improvements in the quality of the methods available to computer architects looking to optimize their designs.

Bibliography

[1] L. Alvarez. “Design Optimization based on Genetic Programming.” PhD thesis, University of Bradford, 2000.

[2] Bates, S. J., Sienz, J., and Langley, D. S. “Formulation of the Audze--Eglais uniform Latin hypercube design of experiments.” *Adv. Eng. Softw.* 34, 8, 493-506. Jun. 2003.

[3] G. Box and N. Draper, *Empirical model-building and response surfaces*. New York: John Wiley & Sons, 1987.

[4] Ceruzzi, P. E. "Moore's Law and Technological Determinism: Reflections on the History of Technology." *Technology and Culture*. Volume 46, Number 3, pp. 584-593. Jul 2005.

[5] S. Eyerman, L. Eeckhout, and K. D. Bosschere. “The shape of the processor design space and its implications for early stage explorations.” In *Proc. 7th WSEAS International Conference on Automatic Control, Modeling and Simulation*, Mar. 2005.

[6] E. Ipek, S.A. McKee, B.R. de Supinski, M. Schulz, R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling." In the *12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XII), San Jose, CA, October 2006.

[7] J. Koza, *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, 1992.

[8] R. Kumar, D. M. Tullsen, N. P. Jouppi, P. Ranganathan. “Heterogeneous Chip Multiprocessors”, *IEEE Computer*, 38(11):32-38, Nov. 2005.

[9] R. Kumar, D. M. Tullsen, and N. P. Jouppi. “Core architecture optimization for heterogeneous chip multiprocessors.” In *Proceedings of the 2006 IEEE/ACM/IFIP International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2006.

[10] R. Kumar, V. Zyuban, and D. Tullsen. “Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling.” In *Proc. 32nd IEEE/ACM International Symposium on Computer Architecture*, pages 408–419, June 2005.

[11] B. Lee and D, Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction." In the *12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XII), San Jose, CA, October 2006.

[12] Y. Li, “Physically Constrained Architecture for Chip-Multiprocessors.” PhD thesis, University of Virginia. 2006.

[13] Y. Li, B. C. Lee, D. Brooks, Z. Hu, and K. Skadron. "CMP Design Space Exploration Subject to Physical Constraints." In *Proceedings of 12th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2006.

[14] K. Madsen, H.B. Nielsen, O. Tingleff. Methods for Non-Linear Least Squares Problems. IMM, Technical University of Denmark. 1999.

[15] Mann, C. C. "The End of Moore's Law?" *Technology Review*. Jan 2002.

[16] H. Neddermeijer, G. von Oortmarssen, N. Piersma, R. Dekker. "A Framework for Response Surface Methodology for Simulation Optimization." In *Proceedings of the Winter Simulation Conference*, 2000.

[17] K. Olukotun and L. Hammond. "Multiprocessors: The future of microprocessors," *ACM Queue*, 3(7): 27-34, September 2005.

[18] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. "Measuring program similarity: Experiments with SPEC CPU benchmark suites.", In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 10–20, Mar. 2005.

[19] Schaller, R.R. "Moore's law: past, present and future." In *IEEE Spectrum*. Volume 34, Issue 6, pp. 52-59. Jun 1997.

[20] L. Schmit and B. Farshi, "Some approximation concepts for structural synthesis." In *AIAA Journal*, Vol. 12, No. 5, pp. 692-699, 1974.

[21] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2000/, 2000.

[22] V. Toropov and L. Alvarez, "Application of genetic programming and response surface methodology to optimization and inverse problems." In *International Symposium on Inverse Problems in Engineering Mechanics* (ISIP'98), Nagano City, Japan, 1998.

[23] V. Toropov and L. Alvarez, "Application of genetic programming and multipoint approximations to optimization problems with random noise." Annual GAMM Meeting, Bremen, Germany, 1998.

[24] V. Toropov, L. Alvarez, and H. Ravaii, "Recognition of damage in steel structures using genetic programming methodology." In *Second International Conference on Identification in Engineering Systems*, Swansea, UK, pp. 382-391, 1999.

[25] G. Venter, R.T. Haftka and J.H. Starnes, "Construction of response surfaces for design optimization applications." AIAA paper 96-4040-CP, In *Proceedings of 6th Symposium on Multidisciplinary Analysis and Optimization*, Bellevue WA. 1996.

[26] J. Yi, D. Lilja, D. Hawkins. "A statistically rigorous approach for improving simulation methodology." *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2003, 281-291.

Appendix A – Genetic Programming Methodology

The goal of this appendix is to provide the reader with additional details about the process of using genetic programming to create an approximation function of a design space, focusing on the actual steps involved in the genetic programming algorithm. For this thesis project, I almost completely followed in the footsteps of L. Alvarez [1], though the implementations of the algorithms described in his dissertation are my own. In this section I will generally assume that the reader is familiar with classic genetic algorithms.

Genetic programming is a technique based on evolutionary biology that can be used to optimize a population of computer programs according to their ability to perform a computational task. In our case, the 'program' is the equation representing the response surface, and the 'task' is to match the set of sample data which was obtained from full-scale simulations. Like a genetic algorithm, genetic programming takes a population of candidate individuals, measures their fitness in a user-specified way, and creates a new generation of individuals out of the fittest individuals from the previous generation.

Genetic programming is a powerful technique because it discovers the functional form of the program or equation completely automatically, which means that in this thesis project I did not ever have to specify the form of the approximation function in advance. The output of the GP is a set of very fit equations, encoded as trees. 'Fit' in this case means that they do a good job of matching the sample data I presented to the GP algorithm.

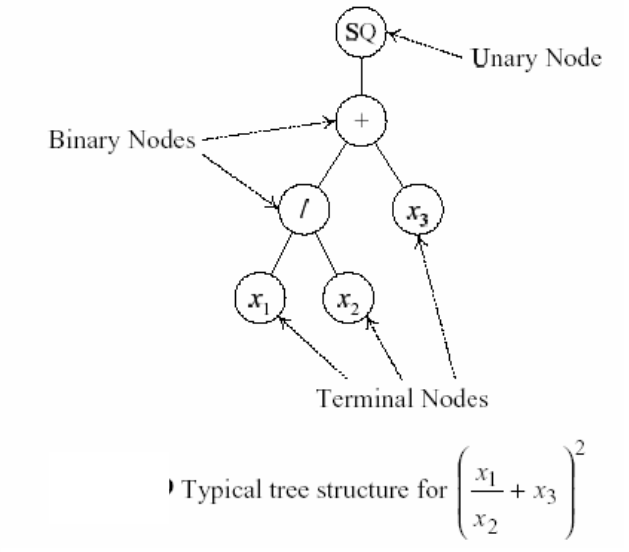

Figure 4. GP tree encoding (from Alvarez [1]).

The individuals in every GP generation are also equations encoded as trees (see Figure 4). These equation trees are initially made up of binary and unary operators (defined by the user), design variables, and eventually also include tuning parameters. For this project, I defined operators for simple arithmetic operations, a few polynomial expressions like square root, and a logarithmic operation. Depending on what kind of expert knowledge the user has, they might easily define trigonometric operators or even more complicated functions if they think such relationships will be useful in capturing the relationships between variables in their target design space. For operators which are undefined for certain inputs (e.g. divide by zero), a harsh fitness penalty is assigned to the tree containing them if such an exception is ever raised during fitness evaluation.

The reproduction process in GP is much like reproduction for genetic algorithms. Two individuals are selected according to their fitness, with fitter trees having a higher probability of being selected, and then swap randomly selected sub-trees (see Figure 5). Mutations are likewise handled in a fashion very similar to that employed by genetic algorithms. A randomly selected node in a randomly selected tree is

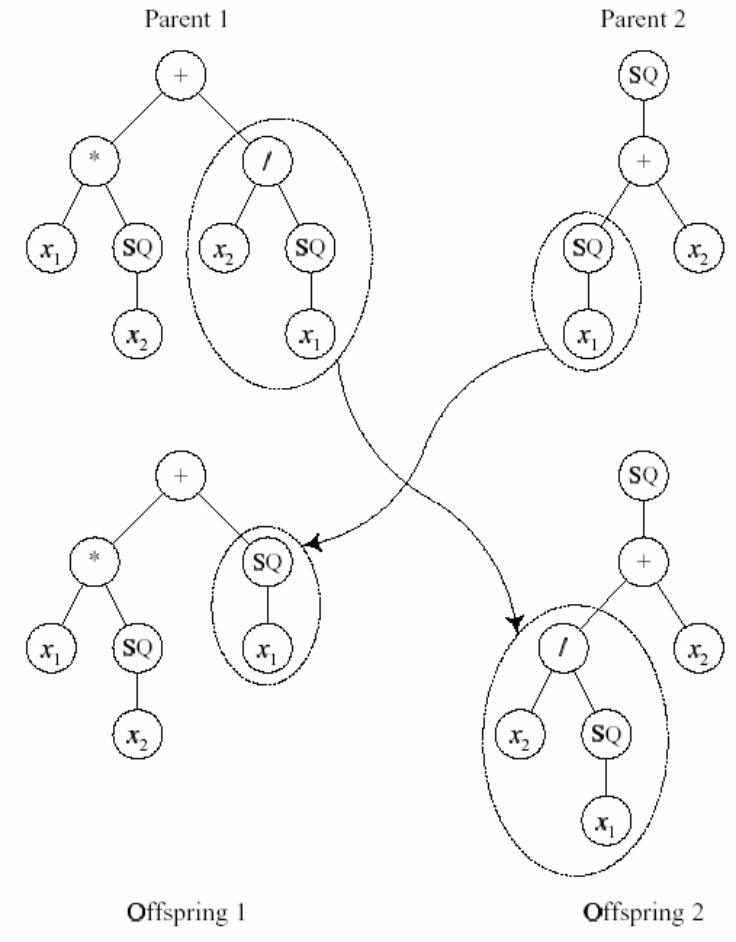

Figure 5. GP crossover operation (from Alvarez [1]).

mutated into a different node of the same type (i.e. operators mutate into other operators, design variables into other design variables).

Two other notable actions are taken during the construction of a new generation of individuals. The first is that a user-specified percentage of unfit individuals from the old generation are culled before even getting a change to reproduce. This acts as a force for convergence, removing the most unfit individuals in order to encourage further breeding between the more successful members of the population. The second action is to transfer a user-specified percentage of the elite, most highly fit individuals directly into the next generation unchanged. These individuals may still breed and produce offspring for the new generation, but transferring them directly ensures that the best candidate solutions found so far can be preserved for multiple generations.

An individual's fitness is evaluated based on two things: the quality of the approximation of the experimental data by the equation tree, and the depth of the tree. Like Alvarez, I estimated the quality of the model by calculating the sum of squares of the difference between the candidate equation's output and the results of runs of the detailed simulation from my design of experiments. Minimizing the depth of the tree serves to keep the resulting equations more compact and straightforward, though obviously care must be taken not to inadvertently promote oversimplification.

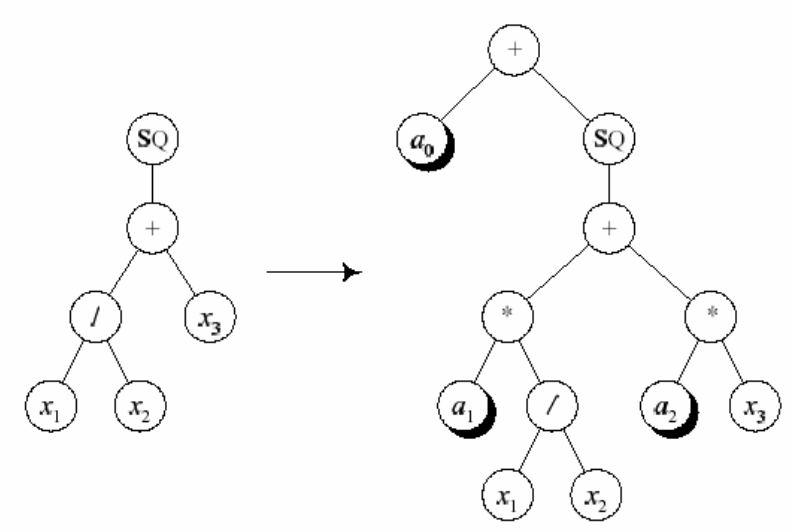

Figure 6. Tuning parameter allocation (from Alvarez [1]).

Because the goal is to find out which equation trees have the best structure, tuning parameters are added to the tree to ensure that every equation is fit to the data as well as possible prior to fitness evaluation.

Tuning parameters are allocated automatically to trees according to their topology and content. Please see Alvarez [1], Chapter 4, for the details of the rules governing tuning parameter allocation. These parameters help to abstract the shape of the approximation function away from the specific value range of the sample data. Once they have been allocated to the tree, the tuning parameters are optimized using both a genetic algorithm and a gradient-descent algorithm [14] in order to try to provide the best possible match for each individual tree to the data. Once all the equations have been tuned, their forms can be compared to one another objectively and fitnesses can be fairly assigned.

The genetic programming process terminates when the fitness of the best individual ceases to improve after a user-specified number of generations, or when a user-specified maximum number of generations have elapsed. In practice I have found that my genetic programming processes tend to converge after no more than 100 generations. As discussed in the conclusion section of this thesis, this result could be due to the experimental parameters or the quality of the gradient-descent algorithm. For more information about genetic programming in general, refer to Koza [7], and for more information about genetic programming as it pertains to the creation of response surfaces, refer to Alvarez [1]. Figure 7 contains a graphical overview of the entire genetic programming algorithm.

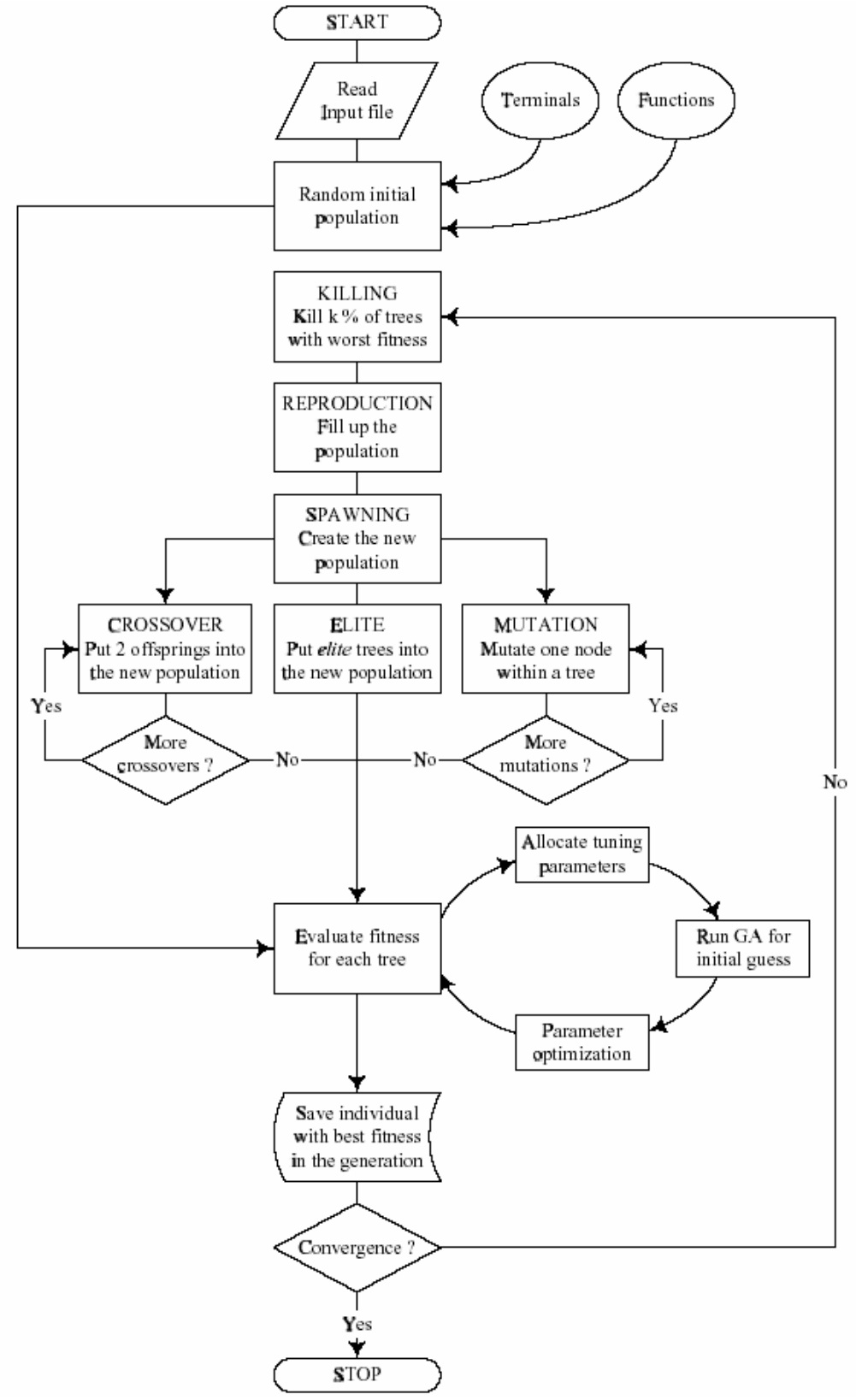

Figure 7. Flowchart of GP algorithm (from Alvarez [1]).

Appendix B – Table of Resulting Equations

SPEC Code	GP response surface equation	Significant design variables
applu	(1.2687 + (-0.532127 * (((89.4726 * (x1 / x11)) - (3.54986e-08 * ((x6 * (x0 / ((x1 / x11) * x0))) * (x8 * x0)))) - (0.00958298 * ((x8 * x11) / x11)))))	x0 = fetch width x1 = frequency x6 = ROB size x8 = LD/ST queue size x11 = L2 cache size
art	(0.30532 + (-0.000495332 * ((94.0503 * x1) + ((-0.000363321 * ((x1 * (x11 / x1)) * (x11 / x1))) - (8.45746e-05 * ((x11 / x1) * (x6 * x8))))))))	x1 = frequency x6 = ROB size x8 = LD/ST queue size x11 = L2 cache size
bzip2	(1.39468 + (0.000206827 * ((0.000199648 * ((x11 * x0) * x11)) + (-820.802 * x1))))	x0 = fetch width x1 = frequency x11 = L2 cache size
crafty	(0.720058 + (-0.00282984 * (((-0.906683 * (x3 * x8)) - (49.6035 * x0)) / x1)))	x0 = fetch width x1 = frequency x3 = branch predictor size x8 = LD/ST queue size
equake	(1.38957 + (0.00043758 * ((-355.743 * x1) + (0.000402084 * (x8 * (((x0 * x6) * x8) / x1))))))	x0 = fetch width x1 = frequency x6 = ROB size x8 = LD/ST queue size
gcc	(1.19834 + (-0.00223846 * ((59.2583 * x1) + (-1.69463 * (x0 * (x4 * x3))))))	x0 = fetch width x1 = frequency x3 = branch predictor size x4 = branch target buffer
mcf	(0.404657 + (-0.000548657 * (((112.978 * x1) - (0.0911179 * (x2 * x8))) - (0.000567741 * (x11 * (x11 / x1))))))	x1 = frequency x2 = max branches x8 = LD/ST queue size x11 = L2 cache size
mesa	(1.25892 + (0.00373005 * ((0.00566059 * ((x0 * x8) * x8)) + (98.9147 * ((x0 / x1) / x1)))))	x0 = fetch width x1 = frequency x8 = LD/ST queue size
mgrid	(0.941694 + (0.00025224 * (((6.14238e-08 * ((x11 * (x8 * x8)) * (x8 * x8))) + (-666.796 * x1)) + (2.88717 * x11))))	x1 = frequency x8 = LD/ST queue size x11 = L2 cache size
swim	(0.106784 + (0.000428321 * ((93.6594 * (x6 / (x1 / (x8 / x6)))) - (82.9734 * (((x8 / x6) / (x1 / (x8 * x6))) / (x8 / (x8 / x6)))))))	x1 = frequency x6 = ROB size x8 = LD/ST queue size
twolf	(1.2072 + (-0.00183701 * ((71.7298 * x1) - (-817.324 * (((x3 / x1) / (x0 * x3)) / x3)))))	x0 = fetch width x1 = frequency x3 = branch predictor size
vortex	(0.658652 + (0.0039152 * (((67.2173 * (x0 / x1)) + (1.87554 * x8)) - (37.3635 * (((x0 / x1) / (x0 / x0)) / ((x0 / x1) / ((((x0 / x1) / x3) / ((x5 / x1) / x0)) / x0)))))))	x0 = fetch width x1 = frequency x3 = branch predictor size x5 = number of ALUs/FPUs

Table 2. Genetically programmed approximation functions and their significant variables (created by author).