This is to certify that the
dissertation entitled

AN EVOLUTIONARY COMPUTATIONAL APPROACH TO

CONFIGURING PORTABLE EMBEDDED SYSTEM

ARCHITECTURES

presented by

JAMES NORTHERN, III

has been accepted towards fulfillment
of the requirements for the

Doctoral degree in Electrical and Computer Engineering

Major Professor's Signature

Nov. 25, 2003

Date

AN EVOLUTIONARY COMPUTATIONAL APPROACH TO CONFIGURING PORTABLE EMBEDDED SYSTEM ARCHITECTURES

By

James Northern, III

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

2003

ABSTRACT

AN EVOLUTIONARY COMPUTATIONAL APPROACH TO CONFIGURING PORTABLE EMBEDDED SYSTEM ARCHITECTURES

By

James Northern, III

Portable embedded systems (*e.g.*, medical equipment, cellular phones, pagers, and video game consoles) are being driven by consumer demands to be thermally efficient (produce less heat), perform faster, and have longer battery life. To design such a system, various hardware units (*e.g.*, level one (L1) and level two (L2) caches, functional units, registers) are selected based on a set of specifications for a particular application. Currently, chip architects are using software tools to manually explore different configurations, so that tradeoffs for consumption, performance, and chip size may be understood.

However, when evaluating multiple design parameters simultaneously, the exploration space expands, design time increases, and human errors become a concern. Genetic algorithms, which are effective in rapid global search of large and poorly understood spaces, have been modified for multiple objectives and applied to guide this process to an improved solution.

This dissertation presents a framework for an evolutionary approach to configuring an "ideal" embedded processor based on power consumption and performance. In addition, a database of simulation results that gives a

more comprehensive evaluation of tradeoffs between power and performance, and of the inter-dependence between parameter configurations is presented. Appropriate search techniques to reduce exploration space and decrease time-to-market are also discussed.

Dedication page

To God, my loving wife, Teasa, and my first born child, Jonah, who have been with me since the beginning and the inspiration for completing this project. To my Mom, brothers, Brian and Winston, and sister, Chiniqua who constantly reminded me of the family motto "Do your best and let God do the rest…".

ACKNOWLEDGEMENTS

I wish to express sincere appreciation to Drs. Barbara O'Kelly and Renate Snider for their assistance in the preparation of this manuscript. In addition, special thanks are due to my committee chair, Dr. Michael Shanblatt, whose critical eye, and enlightened mentoring were instrumental and inspiring. I wish to acknowledge my gratitude to Dr. Erik Goodman, whose familiarity with the subject matter was helpful during the early programming phase of this undertaking. I also thank the members of my committee, Drs. Percy Pierre, Bruce Sagan, and Andrew Mason.

TABLE OF CONTENTS

CHAPTER 3

CHAPTER 4

CHAPTER 5

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Embedded systems are application-specific (single task) computer systems built into larger devices. In today's world, embedded systems are everywhere – homes, offices, cars, factories, hospitals, airplanes, and consumer electronics. An embedded system differs from a general-purpose computer in that it has one fixed application and cannot be changed by the outside environment. However, an embedded system is reactive to internal changes. These types of systems often run in real-time and are designed to meet requirements of low cost and low power consumption, and are usually small in size. In addition, consumer demands have driven embedded systems to be thermally efficient, perform faster, and have longer battery life. Challenging design issues arise, such as multi-objective design goals, configurability (customization), and time-to-market.

This dissertation presents a framework for an evolutionary computational approach to configuring an "ideal" embedded processor based on power consumption and performance. In addition, a database of simulation results that gives a more comprehensive evaluation of tradeoffs between power and performance, and of inter-dependence between parameter configurations (*i.e.*, L1 to L2 cache size, memory bandwidth, instruction window size, datapath width) is presented.

CHAPTER 1

Embedded System Architectures

New portable embedded systems require a high level of performance to meet their hard and/or soft real-time deadlines, where hard real-time is an absolute deterministic response to an event. Examples of embedded systems in the consumer market are digital cameras, video cameras, video game consoles, and medical equipment (*e.g.*, glucose monitor, heart rate monitor). As the complexity and importance of these applications increase, the type of architectures that support them will lead to higher power consumption than traditional portable devices. Thus, limited power-supply capability of current battery technology is forcing designers to explore a combination of high-performance and low-power architectures.

The solution for attaining these new goals is to customize the processor for a particular application. Designers are evaluating three fundamental approaches: RISC ICs, hard cores, and configurable semiconductor intellectual property (IP) cores. Table 1.1 summarizes the three fundamental approaches.

Table 1.1. Fundamental approaches to designing embedded system architectures.

Approach	Description
RISC ICs	Off-the-shelf, stand alone parts that are selected from component data books.
Hard cores	Pre-defined macro layout blocks that can be integrated into application specific ICs.
Configurable semiconductor IPs	Fully programmable processors that can be customized and build by multiple contract semiconductor fabricators.

The advantages of using RISC ICs are their ease of purchase (commodity) and tailored appeal to a specific kind of market. Examples of RISC ICs include Intel's Pentium XScale, IBM's PowerPC, MIPS VR5000, Philips TriMedia TM-1300. Disadvantages of these processors are low integration, poor system performance, and high power requirement. An example of performance and power for selected RISC ICs is shown in Table 1.2.

Table 1.2. Performance and power specifications for RISC ICs.

RISC ICs	Performance	Power
Intel's XScale	1 GHz	1.5 W
IBM's PowerPC 750CX	400 MHz	4.0 W
MIPS VR5000	250 MHz	5.0 W
TriMedia's TM1300	166 MHz	2.7 W

Hard cores are more flexible than RISC ICs because they are pre-defined macro layout blocks that can be integrated into embedded processors. They can be provided and serviced by any application-specific design house. Disadvantages of hard cores include that they are process-specific, that their non-configurable captive IP demands premium price in the market, and that they require additional hardware when integrated into systems-on-a-chip (SoC).

Configurable semiconductor IP cores are more flexible than hard cores because they are not process dependent. They provide foundry-independence for multiple manufacturing sources. The disadvantages of configurable semiconductor IPs are their availability of processor options, configurability of hardware and instruction set architecture (ISA), optimization of software compiler, and integration in an SoC as lack of architectural models of all execution units and peripherals makes it difficult to model paths through the system. In the present work, the configurability of hardware, optimization for power and performance, and ease-of-design (*i.e.*, low design complexity for reduced design time and fast time-to-market) for a configurable semiconductor IP are explored.

1.1 Configurable Semiconductor IP Systems

Configurable semiconductor IP systems are fully programmable processors that can be customized and constructed by multiple contract semiconductor fabricators. These embedded processors are becoming an attractive alternative to RISC ICs and hard cores for the following main reasons: design time and time-to-market are much shorter than that for hardwired custom implementations; time can be reduced if the processor is simulated and explored in software; and the ability to change part of the functionality of any system is crucial in determining the design time. The cost of design and implementation of a programmable processor is lower compared to hardwired implementation. The cost of silicon processing increases in foundries with every new generation of technology [1]. Enabling technologies for programmable processors are becoming mature and competitive. Compilers and design tools are improving at an increasingly rapid rate [2,3] and more attention is being given to improving these tools [4,5]. Programmable processors are now able to provide power versus performance tradeoffs [6,7,8,9,10].

1.1.1 Design Methodology

Many design flows and methods have been proposed in the past, where different steps in design methodology and their consequences on design decisions have been studied. Figure 1.1 shows the different steps involved in the design of a configurable embedded IP system [11]. The five main stages (abstraction levels) in this design methodology are algorithm specification, system-level design, array-level instruction, technology integration, and physical system architecture design [12]. The algorithm specification level is where the type of ISA is chosen. The system-level design stage maps the actual task (C program) to the ISA through a GNU *gcc* compiler at the array-level instruction stage, the processor configuration is explored at the technology integration stage, and the circuit design takes place. The physical architecture system level determines the final layout of the embedded system. The present work addresses the array-level instruction stage where the processor configuration is explored and optimized. In the methodology to be described a genetic algorithm is successfully applied to accomplish this goal.

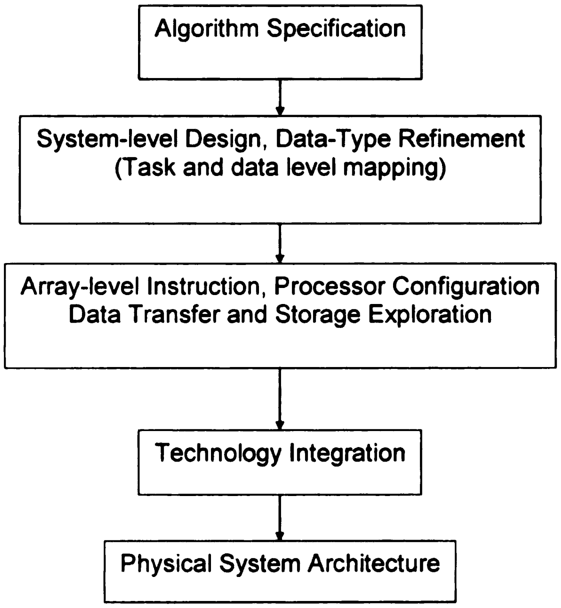
Algorithm Specification
System-level Design, Data-Type Refinement
(Task and data level mapping)
Array-level Instruction, Processor Configuration
Data Transfer and Storage Exploration
Technology Integration
Physical System Architecture

Figure 1.1. System design flow for an embedded system implementation.

1.1.2 Target Architecture

The target architecture consists of basic components in configurable semiconductor IPs. Figure 1.2 shows as an example a general abstraction of the current state-of-the-art multi-media embedded processor [13,14]. The key components typically observed in these architectures are described in Table 1.3.

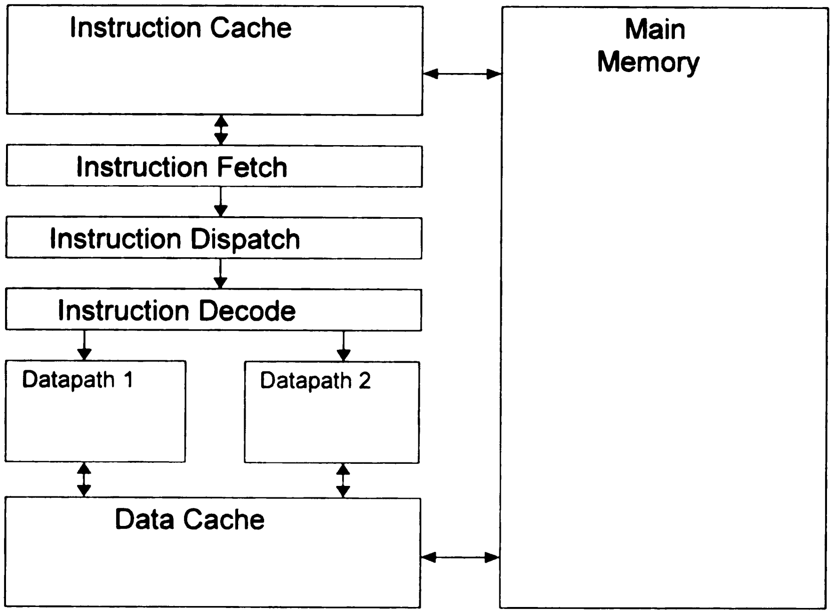

Figure 1.2. Target architecture model for programmable superscalar RISC/VLIW processor with memory hierarchy.

Table 1.3. Basic components of an embedded processor and its key features.

Component	Description
Functional units (FU)	Multiple units for integer and floating-point operations using arithmetic logic unit and multiplier.
Instruction fetch queue	Size and issue width constitute a superscalar [15] or a very long instruction word (VLIW) [13,14] processor or a multi-threaded multiprocessor type of core [16].
Instruction window	Register update unit for the integer and floating-point instructions. Uses a reorder buffer to automatically rename registers.
Cache	One or more levels of cache memories are present, at least one of which resides on chip.
Centralized bus	A centralized bus is selected with one or two data buses, depending on data-dominated multimedia applications [12].

The architecture of embedded programmable processors has features that are parameterizable. Examples include the functional unit, which can vary in datapath size and bitwidths, cache size and organization; the memory unit, which may vary in the number and size of register files; and the interconnections within the processor data paths [17]. The framework for a configurable processor is a compiler coupled to a machine simulator, with parameterizable features tunable by the designer. Interdependence between two parameters has been investigated [18], however, exploration and optimization of all of these parameters have not been addressed.

1.2 Problem Statement

Many embedded processor studies focus on issues related to cache size and organization, and their relationship to energy and power consumption. Although points of diminishing returns exist, decreasing the size generally decreases power consumption. However, the focus of research has shifted to the inter-relationships of other major parameters and their tradeoffs when optimizing for multiple objectives, namely power and performance.

A common difficulty when optimizing for multiple objectives is the conflict between objectives when maximizing or minimizing over a given set of solutions. For example, decreasing cache size reduces power consumption, but also decreases performance. If the cache is too small, cache misses may mask the effects of other processor parameters, such as branch mispredictions, because of parameter interdependency.

Conversely, increasing the number of functional units may increase performance, but also will increase power consumption. While searching the design space for an "ideal" embedded processor configuration, optimizing for two or more criteria can lead to "non-ideal" solutions. Hence, for portable, high-performance embedded implementations, much more advanced techniques, which are efficient in ease-of-design and multi-objective decision-making, need to be explored.

In the context of embedded architecture and organization, the following goals are outlined:

1. Finely tuned performance to meet application requirements: Some embedded designs require high performance for multimedia and communication algorithm processing.
2. Small size: Highly integrated circuits need to be optimized for small size and low system cost.
3. Energy and power efficiency: The embedded processor must operate at a very low MIPS per watt rating to meet requirements for portable applications. For example, a processor's power budget may be a maximum of 1W, but it must still be able to drive the processing for an intense algorithm like speech recognition or data compression.
4. Ease-of-design: Low design complexity will result in reduced design time and fast time-to-market. Design complexity must be kept to a minimum to speed the time-to-market for a system level product. However, when evaluating multiple design parameters simultaneously, the exploration space expands (*e.g.*, four options per parameter with 16 parameters to a set

solution, give 4,294,967,296 possible solutions), thus increasing design time.

1.3 Main Contributions

The main contributions of this work are the following:

1. A methodology for exploring embedded system architectures [Chapter 5]: Issues related to processor architectures and organization, are addressed using an evolutionary approach for configurable processors. An application of a simple genetic algorithm in the exploration of a configurable processor design is used in the process. With this methodology we are able to improve power and performance (*i.e.*, find solution better than original) of embedded processors for practical applications given a large set of parameters [19].

2. A sensitivity analysis for embedded system optimization [Chapter 5]: Based on experimentation with practical applications, we have determined a heuristic for training the genetic algorithm to better represent the configuration problem. The less sensitive parameter value (size, number, *etc.*) is replaced with their best value from a previous GA run, and used as a constant in future evaluations [19]. Therefore, this step reduces the configured design space and refines the exploration criteria.

3. Multi-objective analysis for configuring embedded system architectures [Chapter 6]: An efficient multi-objective genetic algorithm [20] that generates a set of alternative solutions and indicates the best power/performance tradeoff is developed. These alternative solutions are expressed as non-dominated points (*i.e.*, a solution is dominant over another only if it has superior performance in all criteria).

1.4 Structure of Dissertation

The remainder of this dissertation is organized as follows. First, Chapter 2 provides the necessary background material related to this work. An overview of superscalar out-of-order processor simulators is presented. Emphasis is placed on the estimation of power consumption and performance rather than on out-of-order procedures. Also, a methodology for exploring embedded system architectures is presented.

Chapter 3 presents the taxonomy of different search and optimization algorithms. Examples of each method are given and trade-offs between the techniques are reviewed. Based on our research, genetic algorithms (GAs) offer greater flexibility for a larger search space. An explanation of how GAs are used in the configuration methodology is also presented.

Chapter 4 presents the concepts of the GA search and optimization technique. Standard GA concepts such as solution (chromosome) representation, fitness function, reproduction operators, selection criteria, and stopping criteria are discussed. Other techniques for expanding the use of GAs to non-standard problems are also presented.

Chapter 5 presents a methodology for improving power consumption of a computer simulated configurable processor. The functional parameters and the estimation of power consumption are incorporated into a genetic framework. This chapter addresses the use of genetic algorithms and the criteria needed to establish a good search (*i.e.*, population size, random populations, and stopping criteria).

Chapter 6 describes the techniques developed in this work, such as incorporating the parameters into a genetic framework, the sensitivity analysis, and multi-objective decision-making. The encoding and decoding of the parameters are very important in performing an efficient search. The sensitivity analysis involves a technique for training the genetic algorithm to better represent the design search space. A sensitivity analysis of each parameter is performed and the less sensitive values for parameters are substituted for a constant value to reduce exploration time for validation runs. The problems faced with multi-objective decision-making, and the corresponding solutions for power and performance are also discussed. A method is presented for providing a Pareto-optimal front for optimal solutions of two objectives. Experimental results as well as issues related to automation of this technique are presented.

Finally, Chapter 7 presents the main contributions of this thesis and provides directions for future work.

CHAPTER 2

Overview of Simulation Tools

Computer-aided design simulators are used to help in the design of new electronic circuits and devices such as systems-on-a-chip, embedded systems, intellectual property cores. These types of circuits are designed from specifications given at very high levels of abstraction in order to shorten the time-to-market of new products. However, modern processors have become incredibly complex and increasingly hard to evaluate. Architectural simulators have become the solution for evaluating these processors early in the design cycle. An architectural simulator is a tool that reproduces the behavior of a computing device, as illustrated in Figure 2.1.

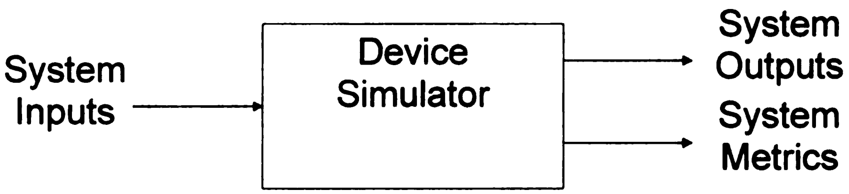

Figure 2.1. High-level view of architectural simulator.

In addition to simulating processors, architectural simulators have advanced to evaluation of multiple designs. The evaluation focuses on the discovery of performance tradeoffs and "ideal" solutions (*i.e.*, better operating configurations within the limited time frame). These tools have been enhanced by the use of search techniques (*e.g.*, simulated annealing, integer linear programming, and genetic algorithms) to aid designers in finding optimal solutions. The work discussed in this chapter provides the background for the simulation and search of optimal designs of portable embedded processors.

2.1 Taxonomy of Simulation Tools

Simulation tools have become an attractive solution to chip architects because more development can be done with software, which allows for greater flexibility and faster leverage in the design cycle. The following benefits are derived from software-based development:

1. Permits more design space exploration.
2. Facilitates validation before commitment to hardware.
3. Level of abstraction (*e.g.*, RTL, compiler, assembly language, VHDL, *etc.*) can be suited for a specific design task.
4. System instrumentation can be increased and/or improved.

To better understand simulation tools and how they work, an example of the taxonomy is shown in Figure 2.2.

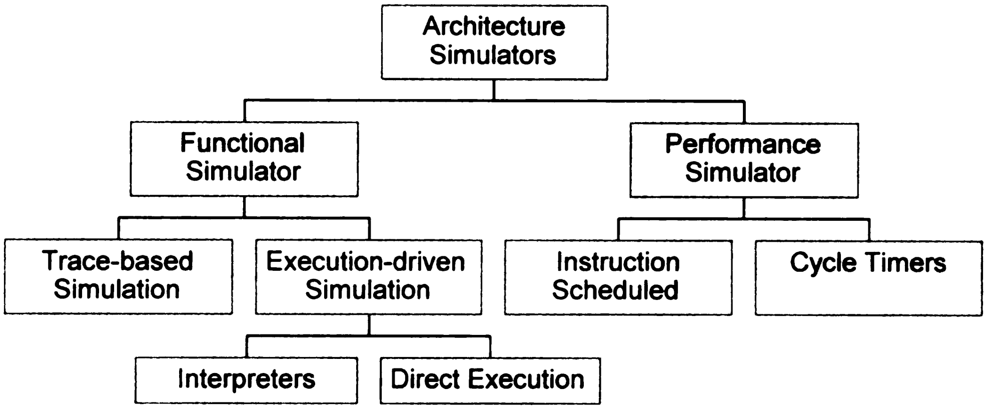

Figure 2.2. Taxonomy of architecture simulators.

The functional simulator implements the instruction set architecture and is concerned with the correctness of the simulation, which is what programmers actually see. It simulates the direct execution of the task. The performance (timing) simulator implements the micro-architecture and models the system internals. This simulator is more concerned with access time and the functions that are not apparent to the programmer such as cache hits and misses.

A functional simulator may be implemented in either an execution- or trace-based manner. Trace-driven simulation reads a "trace" of instructions saved from the previous execution whereas execution-based simulation "runs" the program and generates a stream dynamically. Execution-driven simulation has many advantages, but is more difficult to implement. Instead of implementing direct execution, where an instrumented program runs on a host, an interpreter can be built to accurately simulate execution down to the level of mis-speculated paths.

Performance simulators are based on instruction schedulers or cycle timers. Constraint-based instruction schedulers schedule instructions based on resource availability. Instructions are processed one at a time, in order. They are usually simpler to implement and/or modify, and are generally less detailed. During each cycle, cycle-timer simulators track the micro-architecture state and many instructions in various stages at any time. The simulator state is the same as the micro-architecture state. This type of simulator is good for detailed micro-architecture simulation.

In this thesis, the emphasis will be placed on the SimpleScalar [21] toolset because it is currently a widely accepted tool among researchers. The SimpleScalar toolset uses a combined functional and performance simulator, where instructions are execution-driven and the simulator is synchronized with the micro-architecture state.

2.2 SimpleScalar Toolset Overview

SimpleScalar [21] provides a toolbox of simulation components (a branch predictor module, a cache module, and a statistics-gathering module) as well as several simulators built from these components. Each simulator interprets executables compiled by *gcc* version 2.6.3 for a virtual instruction set (PISA) that most closely resembles MIPS IV [22]. A graphical overview of the toolset is shown in Figure 2.3. Benchmarks written in FORTRAN are converted to C using Bell Lab's *f2c* converter. Both benchmarks written in C and those converted from FORTRAN are compiled using the SimpleScalar version of *gcc*, which generates SimpleScalar assembly code. The SimpleScalar assembler and loader, along with the necessary libraries, produce SimpleScalar executables that can then be fed directly into one of the provided simulators. SimpleScalar optimizes performance and flexibility by reducing design time and maximizing design exploration. In addition, it provides portability for big or little endian machines and varied detailed for different levels of simulations.

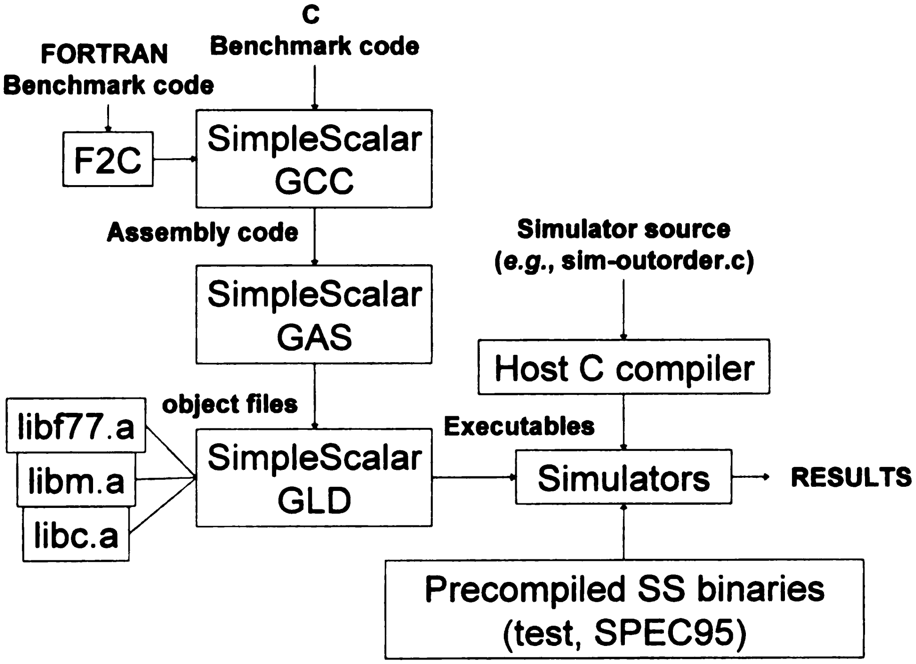

Figure 2.3. Graphical view of SimpleScalar.

2.2.1 Out-of-order Processor Simulator

The most complicated and detailed simulator in the SimpleScalar toolset is *sim-outorder* which supports out-of-order issue and execution instructions. The pipeline for *sim-outorder* is handled in five stages: fetch, dispatch, issue, wirteback, and commit.

Sim-outorder simulates a unified active list, issue queue, and rename register file (register update unit). The register update unit (RUU) handles register synchronization and communication. Entries are allocated at dispatch and deallocated at commit. Using an RUU eliminates artifacts arising from interactions between active list size and issue queue size, and reduces the number of architecture variables to be examined.

The processor's memory system employs a load store queue (LSQ). The LSQ handles memory synchronization and communication. Stored values are placed in the queue if the store is speculative. Loads are dispatched to the memory system only when addresses are known not to conflict. Loads may be satisfied either by the memory system or by an earlier stored value residing in the queue, if their addresses match. Speculative loads may generate cache misses, but a speculative translation look-aside buffer (TLB) misses may stall the pipeline until the branch condition is known.

The five-stage pipeline is illustrated in Figure 2.4. The pipeline is actually traversed backwards, so that inter-stage latch synchronization can be handled correctly with only one pass through each stage.

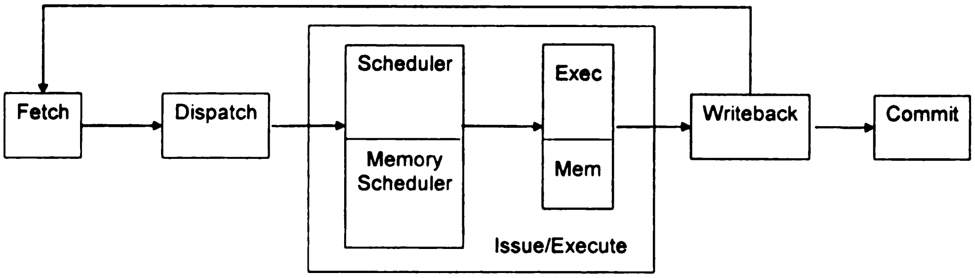

Figure 2.4. *Sim-outorder* five-stage pipeline.

The fetch stage models the machine fetch bandwidth and takes the following inputs: program counter, predictor state, and misprediction from branch execution units. During each clock cycle, this stage fetches the instructions from one instruction cache line and block until I-cache or I-TLB are resolved. After fetching the instructions, it places them in the dispatch queue (IFQ), and probes the line predictor to obtain the correct cache line to access in the next cycle.

The dispatch stage models the processor decode, rename, RUU/LSQ allocation bandwith and architected machine state for execution. During each clock cycle, instructions are decoded and executed, where early detection of branch mispredictions is permitted. If a branch misprediction occurs, stat copy-on-write of architected state to speculative buffers is done. Finally, instructions are entered into the RUU and LSQ, rename table, and machine state.

The issue stage is split into two sub-stages: scheduler and execute sub-stage. The scheduler unit models instruction wake-up, selection, and issue. It separates schedule, track register and memory dependencies. The scheduler uses inputs from the RUU and LSQ. During each cycle, the scheduler locates instructions with all register inputs ready and loads with all memory inputs ready. The issue function then updates the RUU and LSQ and the functional unit state. The execute sub-stage models the functional units and data cache. It accepts as inputs the instructions ready to execute, as issued by the scheduler, and the states of the functional unit and data cache. Each cycle, the execute stage takes the ready instructions, which are supported by issue bandwidth, finds a free functional unit and access port, and reserves the unit for entire issue latency. Finally, the writeback events are scheduled using the operation latency of the functional unit. The execute sub-stage updates the functional unit, data cache state, and writeback events.

The writeback stage models writeback bandwidth and the wake-up ready instructions, detects mispredictions, and initiates misprediction recovery. The function uses completed instructions as indicated by the event queue and RUU/LSQ state for wake-up walks. The event queue is updated during each cycle. When it finds a completed instruction, it walks the dependence chain of the instruction outputs to mark instructions that are dependent on the completed instruction. If a dependent instruction is waiting for that completion, the routine marks the instruction as ready. The writeback stage also detects branch mispredictions. When a misprediction occurs, it rolls the state back to the checkpoint, discarding the erroneously issued instructions.

The commit stage handles the instructions from the writeback stage that are ready to commit. This function does in-order committing of instructions, updating of the data caches (or memory) with stored values, and TLB miss handling. The routine retires instructions at the head of the RUU that are ready to commit until the head instruction is the one that is not ready. When an instruction is committed, its result is placed in a register file and the RUU/LSQ resources devoted to that instruction are reclaimed.

2.2.2 Simulator Command Options

Sim-outorder is a detailed superscalar simulator that runs slower than most of the other simulators in SimpleScalar. *Sim-outorder* provides the user with a host of command-line options for the processor core (Table 2.1), memory hierarchy (Table 2.2), and branch predictor (Table 2.5). Subsequent tables for the memory and branch configuration are shown in Tables 2.3 and 2.6.

Table 2.1. Command-line options for the processor core.

Line Option	**Description**
-fetch:ifqsize <size>	Sets the fetch width to be <size>. Must be a power of 2, and the default is 4.
-fetch:speed <ratio>	Sets the ratio of the front-end speed relative to the execution core (allowing <ratio> times as many instructions to be fetch as decoded per cycle).
-fetch:mplat <cycles>	Sets the branch misprediction latency. The default is 3.
-decode:width <insts>	Sets the decode width to be <insts>, which must be a power of two. The default is 4.
-issue:width <insts>	Sets the maximum issue width in a given cycle. Must be a power of two. The default is 4.
-issue:inorder	Forces the simulator to use in-order issue. The default is False.
-issue:wrongpath	Allows instructions to issue after a misspeculation. The default is True.
-ruu:size <insts>	Sets the capacity of the RUU (in instructions). The default is 16.

Table 2.1. Command-line option for the processor core (cont.).

-lsq:size <insts>	Sets the capacity of the load/store queue (in instructions). The default is 8.
-res:ialu <num>	Specifies number of integer ALUs. The default is 4.
-res:imult <num>	Specifies number of integer multipliers/dividers. The default is 1.
-res:memports <num>	Specifies number of L1 cache ports. The default is 2.
-res:fpalu <num>	Specifies number of floating point ALUs. The default is 4.
-res:fpmult <num>	Specifies number of floating point multipliers/dividers. The default is 1.

Table 2.2. Command-line option for memory hierarchy.

Line Option	**Description**
-cache:dl1 <config>	Configures a level-one data cache.
-cache:dl2 <config>	Configures a level-two data cache.
-cache:il1 <config>	Configures a level-one instruction cache.
-cache:il2 <config>	Configures a level-two instruction cache.
-tlb:dtllb <config>	Configures the data TLB.
-tlb:itlb <config>	Configures the instruction TLB.
-flush <boolean>	Flush all caches on a system call; (<boolean>=0|1|true|TRUE|false|FALSE).
-icompress	Remap SimpleScalar's 64-bit instructions to a 32-bit equivalent in the simulation (*i.e.*, model a machine with 4-word instructions).
-pcstat <stat>	Generate a text-based profile.
-cache:dl1lat <cycles>	Specify the hit latency of the L1 data cache. The default is 1 cycle.
-cache:dl2lat <cycles>	Specify the hit latency of the L2 data cache. The default is 6 cycles.
-cache:il1lat <cycles>	Specify the hit latency of the L1 instruction cache. The default is 1 cycle.
-cache:il2lat <cycles>	Specify the hit latency of the L2 instruction cache. The default is 6 cycles.
-mem:lat <1st> <next>	Specify the main memory access latency (first, rest). The defaults are 18 cycles and 2 cycles.
-mem:width < bytes>	Specify width of memory bus in bytes. The default is 8 bytes.
-tlb:lat <cycles>	Specify latency (in cycles) to service a TLB miss. The default is 30 cycles.

The cache configuration involves the name, number of sets, block size, associativity, and replacement policy. The meaning for each of these fields is listed in Table 2.3. The cache configuration (`<confg>`) is formatted as follows:

```
<name>:<nsets>:<bsize>:<assoc>:<repl>.
```

Table 2.3. Cache configuration fields.

Field Name	Description
<name>	Cache name, must be unique
<nsets>	Number of sets in the cache
<bsize>	Block size (for TLBs, use the page size)
<assoc>	Associativity of the cache (power of two)
<repl>	Replacement policy (l|f|r), where l=LRU, f=FIFO, r=random replac.

The cache size is the product of <nsets>, <bsize>, and <assoc>. To have a unified level two cache in the memory hierarchy, the instruction cache has to be pointed to the name of the data cache in the corresponding level. The defaults used are listed in Table 2.4.

Table 2.4. Default cache configuration.

Name	Configuration	Size
L1 data cache	dl1:256:32:1:l	8 KB
L1 instruction cache	il1:256:32:1:l	8 KB
L2 unified cache	ul2:1024:64:4:l	256 KB
Instruction TLB	Itlb:16:4096:4:l	64 entries
Data TLB	dtlb:32:4096:4:l	128 entries

Choosing the following flag with one of the six subsequent arguments specifies the type of branch prediction (*i.e.*, -bpred <type>). The default mode is a bimodal predictor with 2048 entries.

Table 2.5. Specifying the branch predictor.

Type	Description
nottaken	Always predict not taken.
Taken	Always predict taken.
Perfect	Perfect predictor.
Bimod	Bimodal predictor, using a branch target buffer (BTB) with 2-bit counters.
2lev	2-level adaptive predictor.
comb	Combined predictor (bimodal and 2-level adaptive).

Table 2.6. Predictor-specific command-line options.

Line Option	Description
-bred:bimod <size>	Set the bimodal predictor table size to be <size> entries.
-bpred:2lev <config>	Specify the 2-level adaptive predictor.
-bpred:comb <size>	Set the meta-table size of the combined predictor at <size> entries. The default is 1024.
-bpred:ras <size>	Set the return stack size to <size>. The default is 8.
-bpred:btb <sets> <assoc>	Configure the BTB to have <sets> sets and an associativity of <assoc>. The defaults are 512 sets and an associativity of 4.
-bpred:spec_update <stage>	Allows speculative updates of the branch predictor in the decode or writeback stages (<stage>=[ID|WB]). The default is non-speculative updates in the commit stage.

The SimpleScalar out-of-order issue simulator offers flexibility to create and test different configurations. The values a user can specify and the tests for performance tradeoffs are infinite. Simulation time of *sim-outorder* is dependent upon a particular task or application (*e.g.*, less than 1 sec, without application). In the following section, processor performance and comparison techniques are discussed.

2.2.3 Processor Performance Estimation

In comparing design alternatives, where the user is interested in reducing response time, the chip architect will often relate the performance of two different processors. The response time is the time between the start and completion of an event. Increasing performance implies decreasing execution time. To avoid confusion between increasing and decreasing, we will use the terms "improve performance" or "improve execution time". Execution time can be defined as elapsed time to complete a task. For a processor, CPU time refers to the time the CPU is computing a task or group of tasks, not including I/O or other processes.

Most computers define speed or performance using a clock running at a constant rate. The discrete time events of a clock running at a constant rate are called clock cycles. Chip architects refer to the time of a clock period by its duration (*e.g.*, 2 ns) or by its rate (*e.g.*, 500 MHz). CPU time for a program or task can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a task} * \text{Clock cycle time} \tag{2.1}$$

or

$$\text{CPU time} = \text{CPU clock cycles for a task} \div \text{clock rate.} \tag{2.2}$$

In addition to the number of clock cycles needed to execute a task, a count can be taken of the number of instructions executed – the instruction path length or instruction count (IC). If the number of clock cycles and the instruction count are known, the average number of clock cycles per instruction (CPI) can be calculated as

$$\text{CPI} = \text{CPU clock cycles for a task} \div \text{IC.} \tag{2.3}$$

Therefore CPU execution time is the product of the number of instructions executed, clock cycles per instruction and clock cycle time. Since we are making comparisons of different configurations within the same hardware technology and compiler, the only needed metric for performance is CPI, which is dependent on hardware organization and the instruction set architecture. The performance gain (speedup) that can be obtained by improving some portion of a processor can be calculated using Amdahl's Law [89]. Speedup is the ratio of execution time for the entire task (without using the enhancement) to the execution time for entire task (using the enhancement) and is given by

$$\text{Speedup}_{\text{new conf}} = \text{CPI}_{\text{original}} \div \text{CPI}_{\text{new conf}}. \tag{2.4}$$

SimpleScalar produces calculated results for CPI, IPC and IC along with other processor statistics. From this information, comparison of performance parameters can be made.

2.2.4 Validation of Out-of-order Simulator

There have been four approaches to validating the results produced by *sim-outorder*: micro-benchmark validation, correlation with independent simulators, regression correlation, and code inspection. The SimpleScalar group has run a number of small programs (micro-benchmarks) to test various parts of the machine simulator [23]. For example they compared SimpleScalar ARM model to Intel's StrongARM SA-11XX processors and were within 4% of the real hardware performance. Table 2.7 shows the results of their comparison.

Table 2.7. Validation of performance (CPI) measures via benchmark testing [23].

Benchmark	SimpleScalar	SA-1110	% Difference
cache_hit	1.02	1.01	0.9
cache_miss	33.87	33.70	0.5
br_taken	1.04	1.02	1.9
br_nottaken	1.97	1.91	3.1
bzip2.10	3.20	3.10	3.2
cc1 –O cc1in.i	2.84	2.90	2.1
fft.arm short.pcm	1.45	1.44	0.1

Correlation with multi-scalar simulators, which were developed independently over the SimpleScalar framework, has been done for performance estimation using standard benchmarks [23]. The results were within 5% of the *sim-outorder* simulator. They also compared *sim-outorder* to other published results. However, this was less productive since *sim-outorder* is more detailed than many other dynamically scheduled processor simulators. Regression correlation was done between release versions one and two of *sim-outorder*. Any deviations were tracked down and fixed. Code inspections were run by researchers at University of Wisconsin-Madison and other schools [23]. This procedure has uncovered occasional performance bugs, which has increased the confidence of the SimpleScalar group that the code correctly models a reasonably detailed micro-architecture.

2.3 Wattch Power Simulator Overview

Wattch is an architectural simulation tool built for analyzing and optimizing processor power dissipation [10]. Prior work on architecture-level techniques for power optimization has mainly focused on caches [24,25,26,27]. Historically, two factors led to this focus. For embedded processors, a large portion of their power consumption is budgeted to caches, in some cases up to 40% [28]. Also, since caches are regular structures (*i.e.*, bitline, wordline, sense amplifier, precharge) they are somewhat easier to model and quantify for power studies.

Some work on architectural-level power reduction has been addressed for other areas of the processor [29,30,31]. One of the major shortcomings that was noticed by Brooks *et. al.* [30] in the area of architectural-level power reduction is the lack of a high-level, parameterizable, simulator framework that can accurately quantify power savings [10]. The point of their work was not to compete with lower-level tools but rather to expose the basics of power modeling at a higher level to computer architects and compiler writers. Brooks *et. al.* accomplished this goal by quantifying power consumption of all major units of the processor and integrating these power estimates into a high-level simulator (*e.g.*, SimpleScalar).

Figure 2.5 shows three possible ways Wattch can be used. The leftmost scenario applies to cases where the user is interested in comparing several design configurations that are achievable simply by varying parameters for hardware structures. The middle usage scenario is for software structures involved in compiler development, where a single hardware configuration is used and different compiler programs are simulated and compared. The rightmost scenario highlights Wattch's modularity, where additional hardware modules can be added to the simulator. For our purposes, we chose the first scenario for micro-architectural tradeoffs and exploration. In the following section, the power modeling methodology is described.

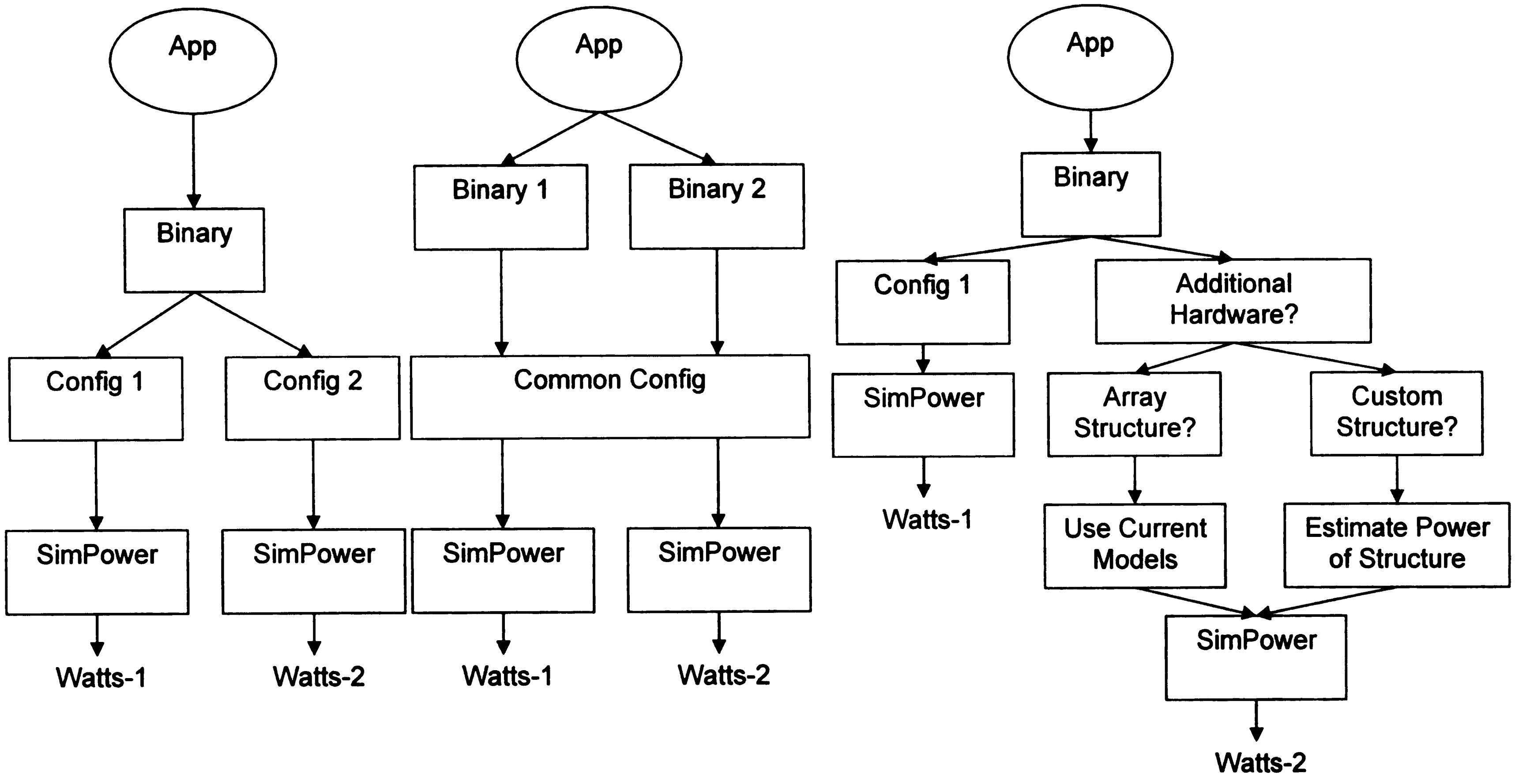

Figure 2.5. Three scenarios for using architecture-level power analysis [30].

2.3.1 Power Modeling Methodolgy

The basis for the power modeling tool is parameterized power models of common structures present in modern embedded processors. The main units that are modeled fall into four categories:

- Array Structures: Data and instruction caches, cache tag arrays, all register files, register alias table, branch predictors, and large portions of the instruction window and LSQ.
- Fully Associative Content-addressable Memories (CAM): Instruction window/reorder buffer wakeup logic, load/store order checks, and TLB's.
- Combination Logic and Wires: Functional units, instruction window selection logic, dependency check logic, and result buses.
- Clocking: Clock buffers, clock wires, and capacitive loads.

Each model estimates capacitance based on the circuit and transistor sizing. In CMOS microprocessors, dynamic power consumption is the main source of power consumption and is defined as

$$P_d = \alpha C V_{DD}^2 f \qquad (2.6)$$

where *C* is the load capacitance, V_{DD} is supply voltage and *f* is the clock frequency. The activity factor, α, is a fraction between 0 and 1 indicating how often, on average clock ticks lead to switching activity. The supply voltage, clock frequency, and load capacitance are derived from 0.35µ technology process parameters used from Palacharla, Jouppi, and Smith [32]. The activity factor is based on the execution of benchmark programs. For circuits that pre-charge and discharge on every cycle (*i.e.*, double-ended array bitlines), $\alpha = 1$ is used. For other sub-circuits, such as single-ended array bitlines, the activity factors are estimated from the benchmark programs using the architectural simulator. For circuits where the internal nodes are unable to estimate switching activity (*e.g.*, decoder and encoder), an assumption of 0.5 for random switching is used. Table 2.8 summarizes capacitance formulas used in the power analysis methodology [30].

Table 2.8. Capacitance formulas used in power analysis methodology [32].

Node	**Capacitance Equation**
Regfile Wordline Capacitance	C_{diff}(WordlineDriver) + C_{gate}(CellAccess) * NumBitlines + C_{metal} * WordlineLength
Regfile Bitline Capacitance	C_{diff}(PreCharge) + C_{diff}(CellAccess) * NumWordlines + C_{metal} * BLLength
CAM Tagline Capacitance	C_{gate}(CompareEn) * NumberTags + C_{diff}(CompareDriver) + C_{metal} * TLLength
CAM Matchline Capacitance	2 * C_{diff}(CompareEn) * TagSize + C_{diff}(MatchPreCharge) + C_{diff}(MatchOR) + C_{metal} * MLLength
ResultBus Capacitance	0.5 * C_{metal} * NumALU * ALUHeight + 0.5 * C_{metal} * RegfileHeight

2.3.2 Array Structures

The array structure power model is parameterized based on the number of rows (entries), columns (width of each entry), and the number of read/write ports. These parameters affect the size and number of decoders, the number of wordlines, and the number of bitlines. The wordline driver and bitline discharge form the bulk of the power consumption in an array structure.

Modeling the power consumption of the wordlines and bitlines requires estimating the total capacitance on both of these lines. The capacitance of the wordlines includes three main components: diffusion capacitance of the wordline driver, the product of the gate capacitance of the cell access transistor and the number of bitlines, and the capacitance of the wordlines' metal wire. The capacitance of the bitline is modeled similarly, where the total capacitance is equal to the diffusion capacitance of the pre-charge transistor, the product of the diffusion capacitance of the cell access transistor and the number of wordlines, and the metal capacitance of the bitline. These capacitance models provide the option of use for single-ended or double-ended bitlines. It is assumed that register file array structures use single-ended bitlines and that cache array structures use double-ended bitlines.

Multiple ports on the array structure have been taken into account for power consumption in three ways. First, additional ports require an additional transistor connection which will add more capacitance on the wordlines. Second, each additional port requires up to two additional bitlines (each bit and its complement), both of which must pre-charge/evaluate on every cycle. Finally, as each core cell becomes larger it will lead to longer word- and bitlines, thus incurring additional wire capacitance.

In the power model, certain transistors are automatically sized based on the model parameters to achieve reasonable delays. For example, the wordline driver is scaled based on the amount of capacitance on the wordlines. Longer wordlines require more capacitance to carry a signal. For other transistors, the sizing is based on the work from Palacharla *et. al.* [32] and Wilton *et. al.* [38].

2.3.3 Content-addressable Memory Structures

The analysis of Content-addressable Memory (CAM) structures is very similar to that of array structures. However, instead of modeling bitlines and wordlines, taglines and matchlines are modeled. The number of rows (number of tags), columns (number of bits per tag to match) and ports on the CAM are taken into account.

The key sizing parameters in the CAM are: the issue/commit width of the matchline (number of match or taglines in each core cell); the instruction window size (CAM's overall height); and the physical register tag size which equals $\log_2$ of instruction window size (CAM's width). The CAM's overall height is each core (RAM) cell multiplied by the instruction window size. The CAM's overall width is the number of bits in the physical register tag, which share a common wide-OR for the final match that signals when the instruction is ready to issue. The wordlines are used to write new tag values into the CAM structure.

2.3.4 Complex Logic Blocks

The complex logic blocks are modeled by the following structures: instruction selection logic (instruction window), dependency check logic (register renaming unit), result buses, and the functional units. The selection logic and dependency check logic is based on the circuit structures modeled in Palacharla *et. al.* [32] and Bishop *et. al.* [33]. The power consumption of the result buses are modeled by estimating the length of the result buses using the same assumptions about functional unit height made by Palacharla *et. al.*

Previous work of Borah *et. al.* [34] and Zimmerman *et. al.* [35] has investigated the power consumption of various functional units. Their results regarding the power numbers are scaled for process and frequency in order to estimate the power consumption of the functional units.

2.3.5 Clocking Network

The clocking network of a high performance microprocessor can be the most significant source of power consumption [30]. Three sources of clock power consumption are considered: global clock metal lines, global clock buffers, and clock loading. The global clock metal lines are modeled as a modified H-tree network in which the global clock signal is routed to all portions of the chip using equivalent length metal wires and buffers in order to reduce clock skew. This method is similar to that used for the Alpha 21264 [36]. The global clock buffers are large transistors that are used to drive the clock signals throughout the processor at a faster rate. The size and number of transistors are estimated similar to the methods of Fair *et. al.* [36] and Bowhill *et. al.* [37]. Explicit and implicit clock loading is considered. Explicit clock loads are the values of the gate capacitances of pre-charge transistors and other nodes that are directly connected to the clock within the units that are modeled. Implicit clock loads include the load on the clock network due to pipeline registers.

The models described are implemented as a C program using the Cacti tool [38] as a starting point. A summary of the major hardware structures and the correlation for the type of model used is given in Table 2.9.

Table 2.9. Common processor hardware structures and the model used by Wattch.

Hardware Structure	**Model Type**
Instruction Cache	Cache Array (2x bitlines)
Wakeup Logic	CAM
Issue Selection Logic	Complex Combinational
Instruction Window	Array/CAM
Branch Predictor	Cache Array (2x bitlines)
Register File	Array (1x bitlines)
Translation Lookaside Buffer	Array/CAM
Load/Store Queue	Array/CAM
Data Cache	Cache Array (2x bitlines)
Integer Functional Units	Complex Combinational
FP Functional Units	Complex Combinational
Global Clock	Clock

2.3.6 Validation of Wattch Simulator

Brooks *et. al.* [10] presents details on the power models and simulator infrastructure required to perform architectural-level power analysis. They verified the power models against industry circuits and found their results to be within 10% for low-level capacitance estimates, and within 10-13% on average for architectural models of tradeoffs between different structures. Baseline configuration models and the configuration of industry processors are shown in Table 2.10.

Table 2.10. Configuration for base and industry models.

Parameter	Base	Alpha	Pentium	MIPS
INT Instruction Window Size		20		16
FP Instruction Window Size		15		16
MEM Instruction Window Size				16
UOP Instruction Window Size	64		20	
INT Physical Register Size		2x80		64
FP Physical Register Size		72		64
UOP Physical Registers Size	32		40	
Memory Order Queue Size	8	32	20	8
Number of Inst. Fetched per Cycle	4	4	3	4
Number of Inst. Decoded per Cycle	4	4	6	4
Number of Inst. Issued per Cycle	4	6	3	4
Number of Inst. Committed per Cycle	4	4	3	4
Number of Integer Funct. Units	5	4	4	3
Number FP Funct. Units	3	2	1	3
L1 Dcache Size	64 K	64 K	8 K	8 K
L1 Dcache Associativity	2-way	2-way	2-way	2-way
L1 Icache Size	64 K	64 K	8 K	8 K
L1 Icache Associativity	2-way	2-way	2-way	2-way
DTLB Size (fully assoc.)	128	128	64	64
ITLB Size (fully assoc.)	128	128	32	64
Local History Table	1024x10	1024x10	NA	NA
Local Predict	1024x2	1024x3	512x4	512x2
Global History Register	10	12	NA	NA
Global Predict	4096x2	4096x2	NA	NA
Choice Predict	4096x2	4096x2	NA	NA
Feature Size	0.35 um	0.35 um	0.35 um	0.35 um
Vdd	3.3 V	2.2 V	3.3 V	3.3 V
MHz	600	600	200	200

Comparisons between modeled and reported power breakdowns of Pentium Pro and Alpha 21264 are shown In Tables 2.11 and 2.12.

Table 2.11. Comparison between modeled and reported total percentage of power breakdowns for the Pentium Pro [30].

Hardware Structure	Model	Intel Data	% Difference
Instruction Fetch	21.00%	22.20%	5.41%
Register Alias Table	4.90%	6.30%	22.22%
Reservation Stations	8.90%	7.90%	12.66%
Reorder Buffer	11.90%	11.10%	7.21%
Integer Exec. Unit	14.60%	14.30%	2.10%
Data Cache Unit	11.50%	11.10%	3.60%
Memory Order Buffer	4.70%	6.30%	25.40%
Floating Point Exec. Unit	8.00%	7.90%	1.27%
Global Clock	10.50%	7.90%	32.91%
Branch Target Buffer	3.80%	4.70%	19.15%

Table 2.12. Comparison between modeled and reported total percentage of power breakdowns for the Alpha 21264 [30].

Hardware Structure	Model	Alpha 21264	% Difference
Caches	15.30%	16.10%	5.23%
Out-of-order Issue Logic	20.60%	19.30%	6.31%
Memory Management Unit	11.70%	8.60%	26.50%
Floating Point Exec. Unit	11.00%	10.80%	1.82%
Integer Exec. Unit	11.00%	10.80%	1.82%
Total Clock Power	30.40%	34.40%	13.16%

The limitations of the models of Brooks *et. al.* [30] are that they do not necessarily model all of the miscellaneous logic present in real processors. The models are scaled and not the actual numbers for process parameters. Also, all of the parameter specifications are not given for each individual processor, which sometimes makes exact correlation to ideal processors difficult.

CHAPTER 3
Overview of Search Algorithms

Search methods are used in many different areas including scientific computing applications and, in our case specifically, for design automation. The objective of a search algorithm is to systematically examine states to find the optimum. The search produces a path from start state to goal state. The output of a search is a unique solution to the problem which represents the best solution among those evaluated.

Mathematically, the state space can be represented as a graph *G*, which may be defined as a pair (*V*, *E*), where *V* is a set of vertices and *E*, the edges, is a set of unordered pairs of elements from *V* [39]. The elements of *V* are denoted v_i and the elements of *E* are denoted e_{ij}, to represent unordered pairs of vertices $\{v_i, v_j\}$. Equivalently, the graph can be thought of as the search space; the vertices are states and the edges are connections between states. In the search space, each state is defined as a set of parameters which are evaluated by the objective function *f(x)*, where the evaluation of the function is achieved through computer computation.

Search algorithms may be roughly classified into three groups: global, global-local, and adaptive [40]. Global search heuristics include random search and mode-seeking methods. Global-local search heuristics include hill-climbing and clustering methods. Adaptive search techniques include single working point methods (*e.g.*, simulated annealing) and converging set methods (*e.g.*, evolutionary computation). The following sections provide a general description of each class of search algorithms.

3.1 Global Search Methods

A global search method is a procedure for constructing a sequence $\{x_k\}$ of points in X that converges to a point at which the global minimum or maximum of some function f is attained or approximated. Global search methods include random search and integer linear programming algorithms. In a random search, points are sampled uniformly in a given search space and the best point found is given as an estimate of the global optimum. This method is seldom used on its own because of its poor efficiency. However, it has become the basis of more sophisticated search algorithms that have a higher probability of converging [41].

The integer linear programming (ILP) method is based on the linear program problem expressed as

$$minimize\ \{c^T x: Ax = b, x \geq 0, x \in Z^n\}, \tag{3.2}$$

where x is the vector of variables to be solved for, A is a matrix of constants, c and b are vectors of constants, and Z^n is the set of n-dimensional integer vectors. The expression $c^T x$ is called the objective function and the equations $Ax = b$ are the constraints. A relaxed form uses the equation $Ax \leq b$, where b is an upper bound for Ax. Most combinatorial and logical restrictions can be modeled through the use of binary variables. The branch and bound technique is an example of a model that uses binary variables.

3.1.1 Branch and Bound Method

The branch and bound technique is an enumeration tree of continuous linear programs [42]. At the root of this tree is the problem expressed in equation 3.2 with the requirement that of *n*-dimensional integer vectors the set is removed. The solution, x, to this root problem will not have all integer components. Some non-integer solution component x_j is chosen and l_j is defined to be the integer part of x_j, $l_j = \lfloor x_j \rfloor$, which infers two sub-problems. The left-child problem has the additional constraint $x_j \leq l_j$, whereas in the right-child problem $x_j \geq l_j + 1$. The branching process can be carried out recursively, where each of the two new problems will produce two more problems that will branch on one of the non-integer components of their solution. Therefore, the enumeration tree is binary. Examples of the branch and bound technique and its application to design automation problems are described in the following paragraphs.

In [43], Hwang used the branch and bound technique where all lower bound options for energy consumption are explored. For each node, an upper and lower bound of energy was calculated. The most promising node for a particular level is the one with minimum energy for that level (lower bound). Each node is annotated with the energy of two partitions containing the finite state machine with datapath states in each partition. The lower bound and upper bound for the energy of each node is calculated, and then the upper bound value is pruned based on lower bound inequalities. Over a set of experiments, an average of 49.2% energy reduction was achieved using the branch and bound technique. This method was compared to simulated annealing, a stochastic algorithm. The performance of the branch and bound technique was slower, but more effective on average by 5% over the simulated annealing method.

Ishihara and Yasuura [44] used ILP to optimize the static voltage scheduling problem for dynamically variable voltage processors. They defined the problem as follows: for a given *task*$_j$ and *mode*$_i$, find x_{ij} which minimizes energy *E* and satisfies the time constraint *T*. The purpose is to clarify the relation between the variety of variable voltages and their effect on energy reduction. The results showed that if the number of variable voltages is increased, the energy consumption is reduced. Another experiment showed that selecting suitable voltages for applications leads to drastic energy reduction even if the number of variable voltages is very small. Both experiments by Ishihara and Yasurra used ILP to optimize for the best combination of variable voltage processors per task based on power consumed.

3.2 Global-Local Search Methods

Global-local search methods are used for estimating the global minimum by finding local minima. The search algorithm is an iterative improvement approach that minimizes the fitness function between sets, examples of such algorithms include the hill-climbing and greedy method. Global-local search algorithms consider only the immediate gain to be made by moving a node.

Certain problems can evolve when using these methods, such as distinguishing between nodes with equal gain and escaping local optima. However, iterative improvement is nearly a universal approach, either as a post-processing refinement to other methods or as a method in itself for global search optimization. In the following sections, examples are presented for these types of methods.

3.2.1 Kernighan-Lin and Fidducia-Mattheyses Method

The Kernighan-Lin (KL) algorithm or min-cut method was designed to improve two-way partitions [45]. This method usually starts with some feasible solution that is iteratively perturbed into another feasible solution, only adopting the perturbation if it improves the cost function. For example in netlist partitioning, the cost function is typically the number of nets per cut and a given move has a gain corresponding to the decrease in cut nets that result from the move. The algorithm generates a sequence of exchanging moves until every module has been moved exactly once, and then adopts the move with the highest total gain. The algorithm terminates when a pass results in zero gain. Since the introduction of the KL method, the algorithm has been improved so that it requires less computation and obtains better results [46,47,48]. Figure 3.1 shows the process flow of the Kernighan - Lin and Fiduccia - Mattheyses (KL-FM) method.

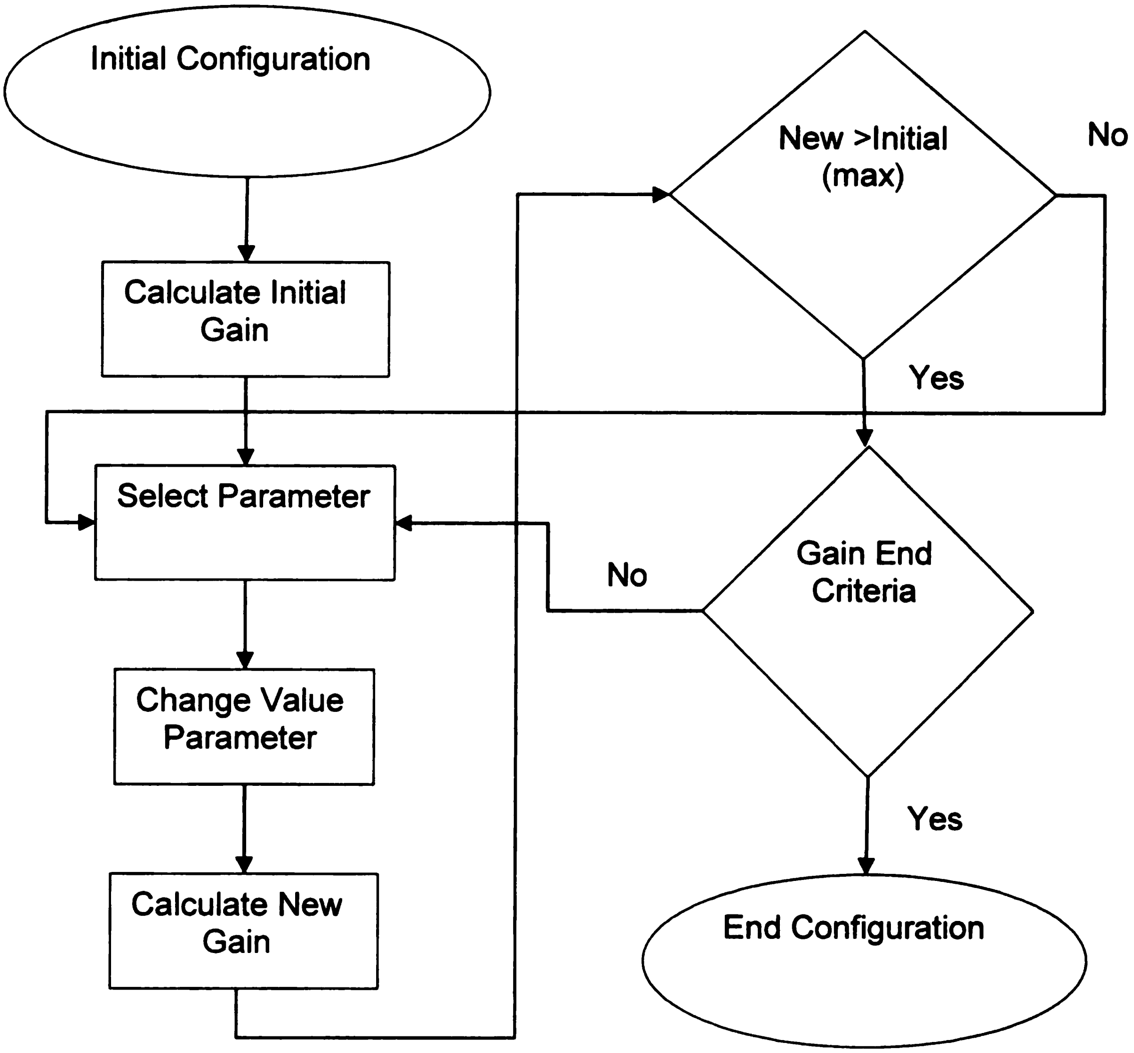

Figure 3.1. Flowchart for KL-FM partitioning algorithm.

Fiduccia and Mattheyses modified the KL algorithm so that it could produce results faster and more efficiently [46]. The key difference between the KL and FM algorithms is that Fiduccia and Mattheyses use a more complicated data structure that allows a single pass through the outer loop to be performed in time proportional to the number of edges in the graph [49]. This difference allows the FM method to achieve a significant reduction in runtime with little loss in solution quality. The time complexity of the KL algorithm is $O(n^2 log(n))$, while that of FM is $O(e)$, where n and e are the number of nodes and edges of G, respectively.

3.2.2 Ratio Cut Method

Wei and Cheng proposed the ratio cut method [50], which is a metric used to locate natural clusters in the circuit and also force the partitions to be of equal sizes. The ratio cost metric for a two-way partition is defined as the sum of weights of the edges cut, divided by the sizes of the two partitions [50]. The size of the partition is equal to the number of nodes within that partition. The method consists of three major phases: initialization, iterative shifting, and group swapping. The ratio cut algorithm partitions a network into small, highly connected groups. These groups form a reduced network. Finally, the FM algorithm is used to improve the reduced network.

3.3 Adaptive Search Method

Adaptive methods are stochastic algorithms that involve probabilistic behavior that is subject to some form of knowledge base. They offer the flexibility of moving in and out of a local minimum. Adaptive search techniques include single working point methods (*e.g.*, simulated annealing) and converging set methods (*e.g.*, evolutionary computation). There are two widely used algorithms in this class: simulated annealing and evolutionary computation including genetic algorithms, genetic programming, and evolutionary algorithms. The simulated annealing algorithm takes an existing solution and then makes successive changes in a series of random moves. Each move is accepted or rejected based on an energy function calculated for each trial configuration. Evolutionary computation simulates the biological process of evolution, where a population of randomly generated candidates evolves toward better solutions by applying operators that are modeled after the natural biological selection process. Many application areas, technical and non-technical, use these algorithms to produce near-optimal solutions. These methods are described in the following section.

3.3.1 Simulated Annealing

Annealing refers to the process used to form glass, iron, *etc.*, into some new shape, by allowing them to cool very gradually from high heat. The goal of the process is to reach the lowest energy state, by moving from higher energy states to lower ones if the cooling process is sufficiently slow, naturally settling in some local minimum. Simulated annealing is a global optimization method that searches between different local optima (maximum and minimum). Starting from an initial point, the algorithm takes a step and the objective function is evaluated. Any downhill step is accepted and the process repeats from this new point. An uphill step may also be conditionally accepted. This uphill decision is evaluated by the Metropolis criteria [51] given as

$$p = exp^{(-\Delta gain/temperature)}. \tag{3.3}$$

As the optimization process proceeds, the length of the steps decline and the algorithm closes in on the search space final solution. The metropolis criteria uses the initial user defined parameters, gain and temperature, to determine the probability of accepting a value of the objective function that is higher. Unlike evolutionary methods in which all parameters are dynamically assigned by the algorithm, the performance of the SA algorithm depends on user defined parameters. This method was originally proposed as a means of finding the equilibrium configuration of a collection of atoms at a given temperature [51]. A flowchart of this algorithm is shown in Figure 3.3.

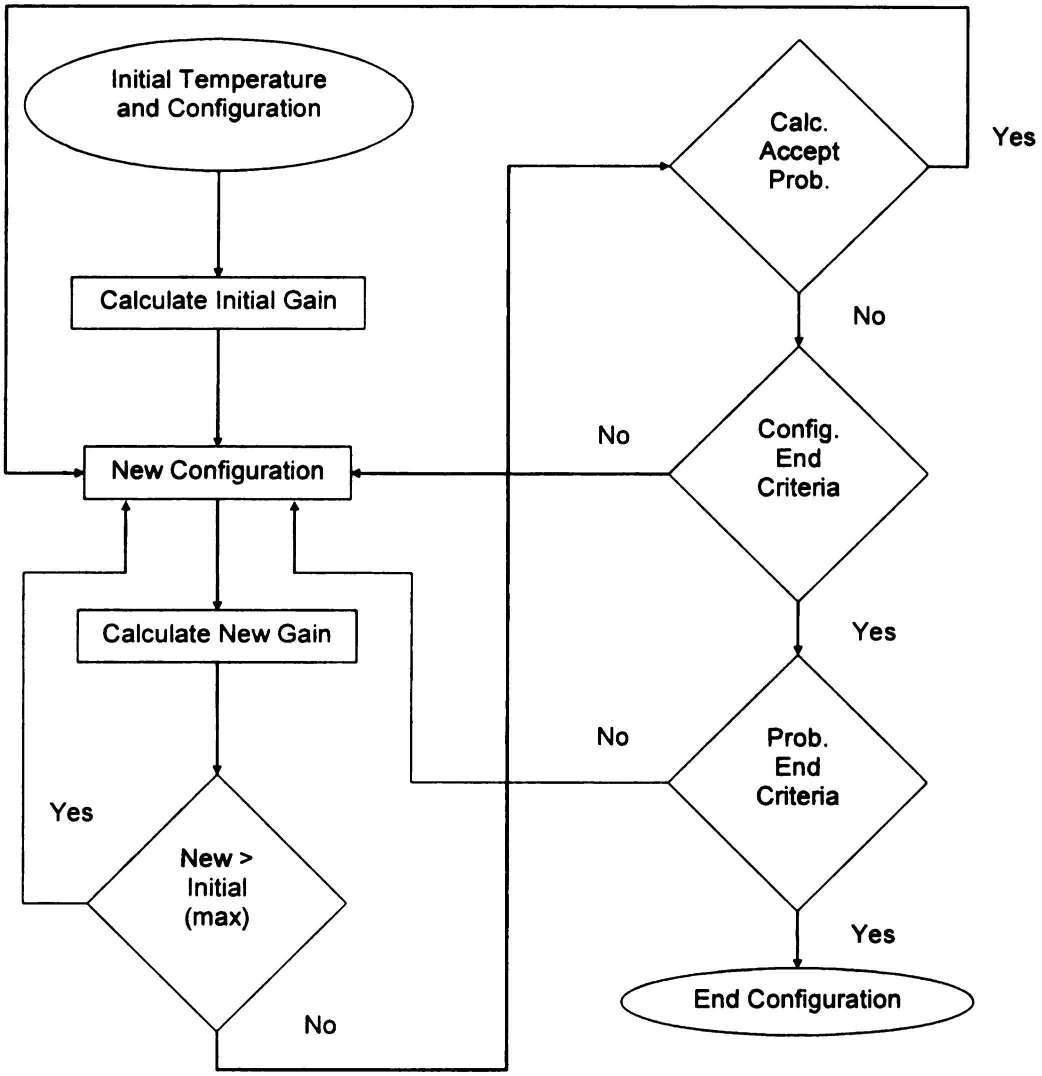

Figure 3.2. Flowchart of the simulated annealing algorithm.

Pincus first noted the relationship between this algorithm and mathematical minimization [52], but Kirkpatrick *et al.* [53] proposed it as the basis of an optimization technique for combinatorial problems [53]. Simulated annealing has a major advantage over group migration methods as it has the ability to avoid becoming trapped in local minima.

Sun and Sechen developed a loosely coupled, parallel simulated annealing algorithm for standard cell placement [54]. The simulated annealing algorithm runs on a standard network of low-cost workstations and yields results similar to the serial version of the algorithm but in much less time. Their parallel algorithm permitted only a small amount of inter-processor communication and used a dynamic region generation scheme. They demonstrated that simulated annealing could be run in parallel, where there was communication between simulations.

Sato's Simulated Quenching (SQ) method is based on simulated annealing and employs the "divide and conquer" technique to give better quality partitions [55]. SQ also has a faster computation time than simulated annealing. SQ was demonstrated as a placement tool that uses partitioning methods such as clustering. Sato's algorithm is based on sorting inside subgroups, where subgroups are generated by cut-lines placed with a constant pitch [55]. The pitch value is decreased step by step from a sufficiently large value to a small value. The pitch value is analogous to the temperature value in simulated annealing.

Two aspects of the simulated annealing process are areas of active research. The first is the number of random configurations and the second is the number of Monte Carlo steps needed at each temperature. If the temperature is decreased too slowly, computation time is wasted. However, if the cooling is too rapid, the search may be trapped in a non-optimal region of search space.

3.3.2 Evolutionary Computing

Evolutionary computing (EC) is based on observation and computer simulation of natural processes in the real world, with the main inspiration stemming from Darwin's "Theory of Evolution" [56]. EC applies these ideas to complex optimization problems and machine learning. The principles of evolution imply that, when organisms are produced, traits found in parents are passed on to their offspring. Variations (mutations), occuring naturally in all species, produce new traits. A process called natural selection tends to favor individuals best adapted to the environment. Over long periods of time, variations can accumulate and produce new species. In natural selection, the fittest survive the longest and produce more offspring. Characteristics encoded in genes are transmitted to offspring and tend to propagate into new generations. In sexual reproduction, the chromosomes of offspring are a mixture of those belonging to the parents.

The search space is a set of all possible encodings of solutions. One measure of the complexity of the problem is the size of the search space. Crossover and mutation implement a pseudo-random walk through the search space. A walk is random because crossover and mutation are non-deterministic. A walk is directed in the sense that the algorithm aims to maximize the quality of solutions using a fitness function. The search process consists of a local and a global search. The local search is looking for solutions near existing solutions in the search space, with crossover as the main operator. The global search looks for solutions with mutation as the main operator for the global search. Figure 3.3 outlines the procedure for a simple evolutionary algorithm.

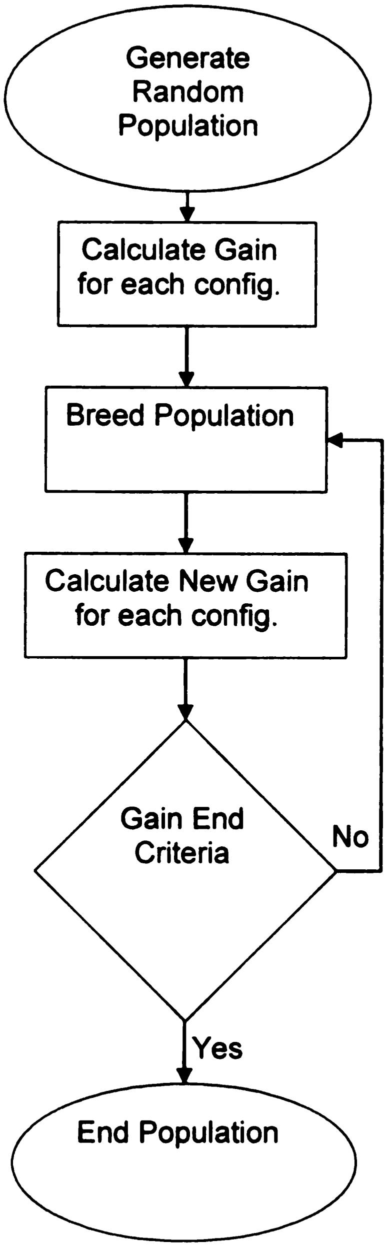
Generate Random Population
Calculate Gain for each config.
Breed Population
Calculate New Gain for each config.
Gain End Criteria
No
Yes
End Population

Figure 3.3. Flowchart of a simple genetic algorithm.

Research in the field of evolutionary computing (EC) has been pursued in the areas of structures undergoing optimization or evolution, reproduction strategy, and genetic operators. Five groups of algorithms have evolved from evolutionary computing: evolutionary programming, evolutionary strategies, classifier systems, genetic algorithms, and genetic programming. Each algorithm differs in how the problem is approached. However, they use the same evolutionary idea. From this group of evolutionary techniques, we chose genetic algorithms as the focus of our study for reasons explained in the next section. Examples of how genetic algorithms are used for semiconductor technologies are provided below.

Fei and Jha used a genetic algorithm in addition to a simulated annealing algorithm to optimize a global system schedule based on system price, power consumption, area constraint violation and real-time constraint violation [57]. A GA-SA was used twice in the methodology, first to allocate cores into an SoC and second to assign tasks to the cores. The output of the multi-level genetic algorithm is a distributed system of SoCs. The trade-off between system price and power consumption was that, as the price increased, the amount of power dissipated decreased.

Martin and Knight used genetic algorithms to optimize simultaneous scheduling and assignment [58]. The GA searches for the best combination of architecture and schedule to minimize the desired function while satisfying the given constraints. Two fields represent each operation. The first field indicates which architecture to use and the second field indicates the time slot in which the operation is scheduled. Transistor count, average power, and peak power were calculated. Peak power optimization achieved a reduction of 47% to 66%, and 70% reduction was achieved by mixing operators of different voltages and combining low power with high speed [58].

3.4 Comparison of Search Methods

A comparison of methods must be based on empirical rather than theoretical evaluation. For example, we can apply a probabilistic method *M* to a problem *P*, in a mapping from (M,P) → (E,m,q), where *E* is the effort applied and *q* is the probability that some minimum *m* is reached. If two methods are applied where the same minimum is achieved then

$$(M_1,P) \rightarrow (E_1,m,q_1)$$
$$(M_2,P) \rightarrow (E_2,m,q_2).$$

For this problem, if less effort is required for the first method ($E_1 < E_2$) and the probability is greater that some minimum is reached ($q_1 > q_2$), then the first method is better than the second method. However, if the same amount of effort is required for the first method ($E_1 = E_2$) but M_1 has a lower probability that some minimum is reached ($q_1 < q_2$), then the second method may be better than the first. In the second case, neither algorithm dominates the other because of a change in probability. Because the results may vary depending on *P* and the levels of *E* and *q*, a conclusive decision about the superiority of one method over another requires excessive computations. Furthermore, if *m* also varies, and one method obtains a better solution, with much smaller probability and the same effort, then only by experimentation may superiority be challenged.

For example, Bright [59] chose two techniques gradient search (hill-climbing) and simulated annealing, which are widely used to solve VLSI design problems, to compare to GAs. Bright's designs created by gradient search were improved by 30% using a GA tool that optimizes for power and area. It was discovered that the best design from the GA tool required an initial design with higher power consumption than the initial design for the gradient search. By using the gradient search method, such designs were prevented from surviving during the process, hence localizing the search space. As for comparing the GA and simulated annealing techniques, both methods produced design results within the same order-of-magnitude in optimizing for power consumption. However, the GA provided flexibility for evaluating a set of solutions for convergence to an optimal solution, rather than depending on single solutions.

CHAPTER 4

Overview of Genetic Algorithms

Genetic algorithms (GA) are a part of evolutionary computing and are especially useful for manipulating large amounts of data. Genetic algorithms are inspired by Darwin's theory of evolution. Rechenberg [60] first introduced the idea of evolutionary computing in 1973, and Holland [61] extended the idea of evolutionary programming in 1975 by developing GAs. GAs encode a potential solution to a specific problem on a simple chromosome-like data structure and apply crossover and mutation operators to these structures to preserve critical information [62].

A genetic algorithm in the case of design automation, randomly selects a population of designs. Each potential design solution is then evaluated for "fitness", which is a measure of relative merit with respect to a defined criterion. The design solutions are combined using crossover probabilities (*e.g.*, from a population of 50, solutions are paired to make a new population of 50). Next, characteristics of a solution are randomly mutated. This new design solution has characteristics of each parent. Usually, the "best fit" characteristics of the design tend to be passed on to the next generation. As a result, each generation of the design builds on the successes of past generations to approach a desired solution. Genetic algorithms are usually applied to spaces that are too large to be exhaustively explored.

4.1 Defining Genetic Algorithm Criteria

Certain areas such as solution representation, fitness function, reproduction operators, and selection criteria must be addressed when defining the criteria for a genetic algorithm. There are many variations to the traditional GA operators and components described in this section, all of which are aimed at improving the efficiency and success of the GA [71]. This chapter serves as an introduction to the fundamental concepts of GAs.

4.1.1 Solution Representation

In order to implement the processor configuration problem using a GA, candidate solutions must be encoded into a chromosome-like structure suitable for manipulation. Traditionally, solutions have been represented by binary-strings [61]. The individual elements within the chromosome, such as each 1 or 0, are known as genes. Subsequent research has developed complex representations such as alphabet-strings [63] and decision trees [64]. The important aspect is that the representation encodes the properties of the solution such that they can be fully explored by the GA.

The choice of chromosome representation is very important if the GA is to be fully exploited. If an unsuitable choice of chromosome representation is used, it could place an unnecessary computational strain on the GA, requiring complex manipulation and decoding for quality evaluation. This affects the performance of the GA in searching the solution space. An unsuitable representation may also prevent the GA from determining an optimal solution, with the chromosome unable to represent all possible solutions or not allow certain operations.

4.1.2 Fitness Function

A fitness value for each solution of the population is evaluated rather than derived from previous information. The fitness is a means of determining the relative quality of each solution. The quality of a chromosome is dependent upon the decoding process and the calculation of its relevant parameters. The fitness is a direct measure of how well the parameters of the encoded solution satisfies the objective function, where the objective function is the goal or desired state of the optimization process.

4.1.3 Reproduction Operators

The reproduction step of the GA selects individuals from the current generation to produce offspring that will enter the next generation. The production of offspring involves the modification and combination of the genes of solutions in the current generation. This is the building block of the search space, as the next generation will consist of a different set of chromosomes comprised from sub-blocks of the previous generation. The chromosomes are modified through the application of genetic operators. The two most common genetic operators are mutation and crossover.

Mutation operates on a single chromosome to modify its characteristics through random manipulation of its genes. An example of the mutation process is illustrated in Figure 4.1 for a binary-string chromosome.

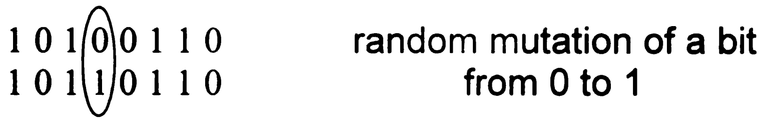

Figure 4.1. Example of a mutation operation.

Mutation is a random feature in the genetic process. For example, for each digit in an individual, a random number R is generated between 0 and 1. If R is smaller than a preset mutation probability, that digit is replaced by its complement, thus generating a mutation.

This example illustrates the random change of the value of a gene within the chromosome. The mutation has produced a new chromosome with characteristics that differ from the parent chromosome. Mutation is primarily used to introduce diversity into the population and encourage the GA to explore new areas of the solution space.

Crossover operates on two chromosomes, combining their genetic material to produce offspring (child) chromosomes. For example, the simplest form of crossover (one-point crossover) proceeds as follows. First, the entire population is paired at random to give N/2 sets of potential parents. Second, pairs of solutions are chosen to crossover with probability P_c. If the generated random number R (between 0 and 1) is smaller than the preset crossover probability, then two new solutions are created by exchanging all the bits following a randomly selected locus on the strings. Figure 4.2 illustrates the example, where crossover after position 5 is proposed between solutions.

	Parent Chromosomes		Child Chromosomes	
Chromosome 1	1 0 1 0 0	1 1 0	1 0 1 0 0	1 1 1
Chromosome 2	1 1 0 1 0	1 1 1	1 1 0 1 0	1 1 0

Figure 4.2. Example of a crossover operation.

A slightly more complex operator, first proposed in Cavicchio [65], is a two point crossover in which two crossover points are randomly selected and the substrings between and including those positions are exchanged. Strings may also be treated as continuous rings. For example in Figure 4.3, if crossover between points 6 and 2 is proposed for strings, then digits from 6 to 8 are switched first, and digits 1 and 2 are switched last. Or vice versa, if crossover is between points 2 and 6, the offspring are from points 1 and 2 switching first, then points 6 through 8 are switched last.

	Parent Chromosomes			Child Chromosomes		
Chromosome 1	1 0	1 0 0	1 1 0	1 1	1 0 0	1 1 1
Chromosome 2	1 1	0 1 0	1 1 1	1 0	0 1 0	1 1 0

Figure 4.3. Example of two-point crossover.

Many other crossover methods exist, including multiple point, masked selection, order-based, splicing of complete tree-subsets [63,66,67,68,69]. However, it has been shown that as more substrings are swapped, performance is degraded. Although it is essential to introduce some changes in order to make progress, too many alterations make the probability of destroying the good features of a solution unacceptably high. An effective GA search requires a balance between exploitation of good features in existing solutions and exploration or combination of introducing new features.

Crossover and mutation are usually applied in a GA with probabilities tailored to suit a specific problem. Crossover applied without an associated mutation operator may prevent exploration of certain regions of the solution space. If only crossover is used, it can only combine information that is already present in the chromosome, which may lead to a single gene having the same value in all chromosomes. Without a mutation operation that gene would never be changed, consequently blocking off a region of the solution space from the search [59].

4.1.4 Selection Schemes

The selection procedure is a key component of the search process. The evolutionary theory of natural selection is based on the idea of 'survival of the fittest', where individuals that are more successful within their environment have a greater chance of reproducing and propagating their characteristics to the next generation [56]. Within the context of a GA, the procedure dictates that individuals that are more successful in meeting the specified objective will have a higher probability of being chosen for reproduction. Thus, better solutions are found by building on the current best solutions. This guides the search procedure to those areas within the search space that contain the best solutions.

Since the initial development of GAs, many techniques have been developed that aim to improve and build on this standard idea [62,70,71,72]. However, the basic principle behind all of these techniques is the probabilistic selection of individuals based on their quality. The Fitness Proportionate Selection (FPS) method is the most commonly used probabilistic selection technique [59].

The FPS method was introduced by Holland based on his analysis of the "2-armed-bandit problem" [63,73]. FPS allocates to each individual a fixed probability of being selected, based upon its fitness relative to the total fitness of all individuals within the generation. For example, let f_i be the fitness value of individual i and let $f_{average}$ be the average population fitness, where

$$f_{average} = (1 / N)(\mathrm{Sum}(1,N)f_i).$$

The probability of an individual being selected is

$$p_i = f_i / (\mathrm{Sum}(1,N)f_i),$$

$$p_i = (1 / N) * (f_i / f_{average}).$$

FPS can be implemented with the "roulette wheel algorithm". A wheel is constructed with markers corresponding to fitness values. For each fitness value f_i, the size of the marker (*i.e.*, the proportion of the wheel's circumference) associated with f_i, is given by p_i. Thus, when the wheel is spun, the probability of landing on f_i is p_i. There are two main ways of simulating the roulette wheel algorithm: vector and cumulative distribution.

Vector representation begins with a vector *v* of *M* elements from {1, ..., *N*} that is constructed so that each subsequent *i* in {1, ..., *N*} has $M*p_i$ entries in *v*. A random index from {1, ..., *M*} is selected and individual *v(r)* is selected. For example, if f_1=f_2=10, f_3=15 and f_4=25, then with *M*=12, *v*=(1,1,2,2,3,3,3,4,4,4,4,4). If *r*=6, then individual *v*(6)=3 is selected. Cumulative distribution representation is where a random real-valued number *r* in {0,(Sum(1,*N*)f_j)} is chosen and individual *i*, such that Sum(1,*i*-1)$f_j \leq r >$ Sum(1,*i*)f_j. Note that by convention Sum(1,0)f_j = 0. Cumulative distribution is effective, but relatively inefficient. Vector representation is efficient, but its effectiveness depends on *M*, (*i.e.*, the value of *M* determines the quantization of the p_i's and thus the accuracy depends on *M*).

The overall problem with FPS is that individuals with above-average fitness tend to have more than one copy in the mating pool, while individuals with below-average fitness tend not to be copied into the next generation. This leads to premature convergence and stagnation. Both problems can be solved using a fitness scaling technique which is described in Section 4.2. However, the advantage of the FPS roulette wheel method is that all solutions have some chance of being selected.

4.1.5 Stopping Criteria

The GA is used to determine the “best” or “ideal” solution to a given problem, which is the global optimum point in the solution space. While a GA cannot be guaranteed to find the global optimum, if implemented correctly it will find a “good” or satisfying solution in a reasonable amount of time. In this case the stopping criterion is specified in terms of the number of generations or solution evaluations. The actual number of generations is usually set very high to increase the likelihood that the GA has settled on a satisfying point in the solution space.

Assessing the convergence of the population is another technique used to determine whether a GA has reached a stopping point. For example, in analyzing every solution in the population, if 95% of the genes in each chromosome are identical in all solutions, then the GA is considered to have converged and the search is terminated [72].

4.2 Non-standard Genetic Algorithms

The previous section introduced the concept of a traditional GA, comprised of standard GA components, which were proposed by Holland (1972) and his successors. Many researchers using GAs to solve complex, real-world engineering problems have proposed that, if GAs are to achieve their full potential in engineering design, it is necessary to specifically design the GA to suit the problem [70,72,74]. Their work leads to the development of non-standard GAs, implemented in a standard genetic framework. The non-standard GA exploits the use of non-standard chromosome representation (*e.g.*, non-binary chromosome, alphabet strings, tree representation, *etc.*), adaptive operator rates, and non-standard genetic operators. Non-standard genetic operators incorporate problem-specific techniques to modify the chromosome according to standard design rules. Incorporation of these rules has been shown to improve the efficiency and results of the GA-based search technique.

The framework for our non-standard GA is based on the Genetic Algorithm Optimized for Portability and Parallelism System (GALOPPS) 3.2.4, developed by Goodman [75]. GALOPPS is a distant descendant of SGA-C, v1.1, with modifications by Earickson, which was based on SGA-C, by Smith, which was based on SGA (in Pascal), and copyrighted by Goldberg in 1986 [75]. GALOPPS is a flexible, generic genetic algorithm that is based on Goldberg's Simple Genetic Algorithm (SGA). GALOPPS extends the SGA by including several different methods for generating solutions via genetic processing. In the following section we describe some of the techniques used in GALOPPS 3.2.4.

4.2.1 Representation of Non-binary Chromosomes

In GALOPPS 3.2.4, the chromosomes may represent different alphabet sizes (cardinalities) for different fields. They can be handled automatically when crossover is applied at field boundaries. Mutations can occur anywhere within the chromosome never to produce non-permissible values. An alphabet refers to a field instead of a bit and each alphabet may have different lengths, meaning different values for a particular parameter (*e.g.*, cache associativity = {1,2,4,8}). GALOPPS automatically decodes the chromosome properly, remaps variable-length fields under an inversion operation. It transforms migrants into the correct inversion pattern (and field boundaries) for the receiving subpopulation.

4.2.2 Scaling of Fitness Function

Fitness scaling offers a way to alleviate premature convergence and stagnation. GALOPPS includes three scaling methods: window scaling, linear scaling, and sigma truncation. For window scaling, fitness values are scaled by either subtraction or division so that the worst value is close to 0 and the best value is close to a certain value, typically 2. Problems arise when the original maximum is very extreme (super-fit) or when the original minimum is very extreme (super-unfit).

Linear scaling is where the fitness f is replaced by a scaled fitness $f' = a*f + b$, where a and b are chosen so that the scaled average is the same as the raw average. The maximum scaled fitness is the number of expected copies desired for the best individual multiplied by the raw average fitness. One problem with linear scaling is that the scaled fitness function may take on negative values if there are a few bad individuals with fitness much lower than the average fitness and fitness close to the maximum fitness. One solution is to arbitrarily assign the value 0 to all negative fitness values. Another solution is to use sigma truncation.

With sigma truncation, the fitness is replaced by the scaled fitness

$$f' = f - (f_{average} - c*sigma)$$

where *sigma* is the population standard deviation, *c* is a reasonable multiple of *sigma* (usually $1 \leq c \leq 3$). Negative results are arbitrarily set to 0. Sigma truncation removes the problem of scaling to values. Truncated fitness values may also be scaled if desired.

4.2.3 Stochastic Universal Sampling

Stochastic Universal Sampling (SUS) is a selection technique introduced by Baker [76]. Baker shows that SUS has minimum spread and zero bias. Bias is an expression for the absolute difference between the expected and the actual number of individuals. Spread is the set of possible numbers of individuals that can be selected by a given sampling strategy. Minimum spread is defined as the smallest spread that allows zero bias. Baker effectively minimizes the genetic drift caused by selection. SUS can be visualized as a "wheel of fortune" with *n* arrows instead of one, with equal angle distance to each other, *n* being the number of individuals in the population.

Non-standard genetic operators incorporate problem-specific techniques to modify the chromosomes according to standard design rules. Incorporation of these rules has been shown to improve the efficiency and results of the GA-based search technique. The framework for our non-standard GA, GALOPPS, extends the simple genetic algorithm by including several different methods for generating solutions via genetic processing.

CHAPTER 5

Embedded Processor Configuration Methodology

Many embedded processor studies currently focus on issues related to cache size and organization, their relationship to energy and power consumption, and datapath width and their effect on performance. Although points of diminishing return exist, such as decreasing the size of one parameter. However, the focus of research has shifted to simultaneously evaluating multiple parameters and their tradeoffs when optimizing for multiple objectives, namely power and performance.

A common difficulty with optimizing for multiple objectives is the conflict between objectives when maximizing or minimizing over a given set of solutions. For example, decreasing cache size reduces power consumption, but also decreases performance. If the cache is too small, cache misses may mask the effects of other processor parameters such as branch mispredictions due to parameter interdependency. Conversely, increasing the number of functional units may increase performance, but also will increase power consumption. While searching the design space for an "ideal" embedded processor configuration, optimizing for two or more criteria can lead to "non-ideal" solutions. An "ideal" embedded processor is defined as one with low power consumption and high performance. Hence, for portable, high-performance embedded implementations, much more advanced techniques, which are efficient in ease-of-design and multi-objective decision-making, need to be developed.

From a system level, using a programmable embedded processor can improve designer productivity and intellectual property reuse. A key to the success of these platforms is that they are heavily parameterized, so that designers can configure the platforms to the particular application, where power, performance, and chip area are improved [78]. The configure-and-execute paradigm proposed by Vahid and Givargis [78] for embedded processors has actually existed for many years in microcontroller-based system design [78]. A microcontroller is an earlier form of embedded processor that has a core processor and peripheral devices (*e.g.*, memory, timers and UARTs) on chip and pre-integrated. Similar to an embedded processor, a microcontroller may be applied to different classes of devices (*e.g.*, televisions, automobiles, personal digital assistants, computerized robots), each class having different peripherals.

The parameterizable components of embedded programmable processors include the functional unit, which can vary in datapath size and bitwidth, cache size and organization; the memory unit, which may vary in number and size of register files; and the interconnections within the processor data paths [17]. Three steps are used to optimize embedded system designs based on components that are parametizable: application, parameter optimization, and new silicon generation. The approach is outlined in Figure 5.1.

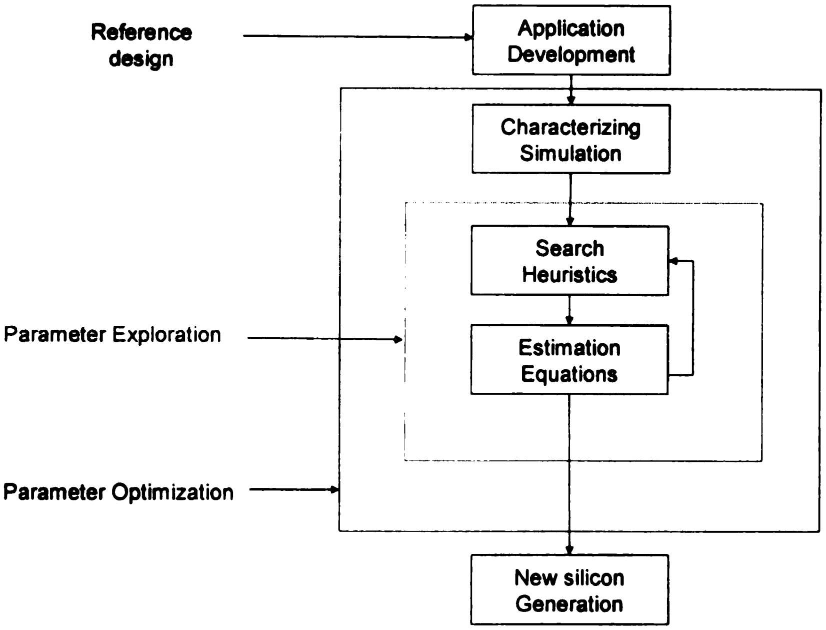

Figure 5.1. Flowchart of parameterized system design.

Application development begins with an available "reference design" [77]. This reference design is implemented on a configurable prototype processor, which is the virtual instruction set architecture. Application development involves the mapping of the task to the virtual instruction set architecture. After application development, parameters are optimized for that particular application based on power, performance, and size optimization heuristics.

Parameters may include:

- bus size, address, and data encoding techniques;
- cache size, associativity, block size/line size, and write-back techniques;
- datapath parameters relating to specific peripheral cores, like buffer sizes;
- resolutions, compression level.

Once the parameters have been optimized, a new chip design is generated based on the optimized architecture. This is an "ideal" chip, which is correct on the first pass because of extensive, previously performed, in-circuit emulation.

Givargis and Vahid [78] used a Pareto-optimal approach to tune for the specified type of architecture with respect to power and performance. All configurations are evaluated for power and performance, then sorted in decreasing order of execution time. An enumerative method of traveling through the search space is used to eliminate all designs that result in power consumption above the derived minimum. The problem can become impractical if the configuration space expands. GAs deal with a converging set of possible solutions for multi-dimensional problems, which leads to a set of "good" individuals in a single run of the algorithm.

5.1 Defining Genetic Algorithm Criteria

The target architecture consists of a processor core, L1 cache, and L2 cache, as illustrated in Figure 5.2. The parameters explored during the configuration process include the issue rate, reorder buffer size, number of floating point ALUs, number of floating point multipliers, number of integer ALUs, number of integer multipliers, load-store queue size, first-level and second-level cache size, block size, associativity, and bus size. The mapping of the genome is shown in Figure 5.3. The italicized blocks represent the specified parameters varied by the GA, and the non-italicized blocks represent the level of hierarchy in the embedded processor.

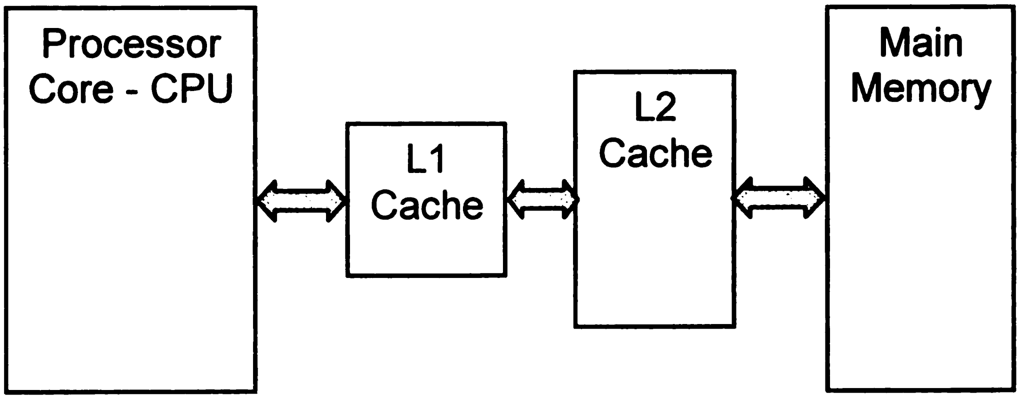

Figure 5.2. Target architecture for embedded system design.

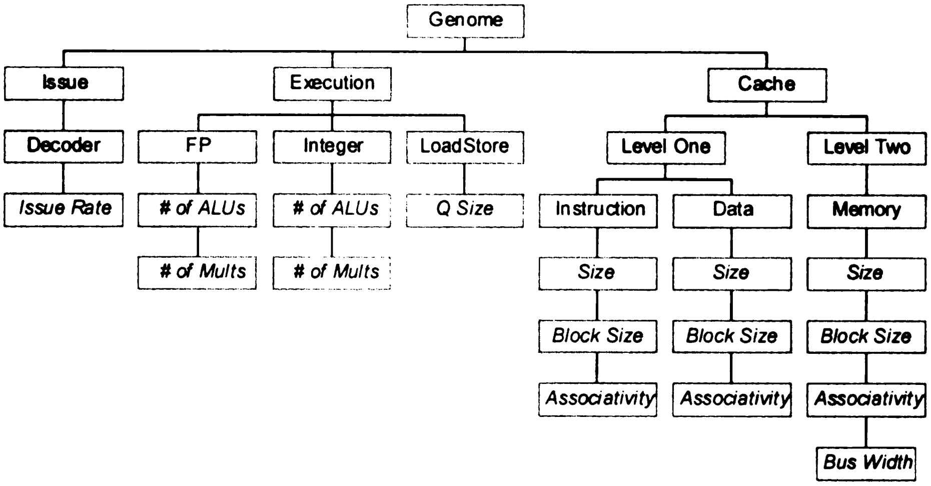

Figure 5.3. Genome structure for GA.

For the purpose of this study, the set of experiments include the 16 different parameters, shown in Figure 5.3, with at least four options per parameter, given in Table 5.1. Exhaustively searching the entire design space for an optimum using an enumerative process would require 4,294,967,296 different configurations, which could take years to complete. On the other hand, if different parameter sets were randomly selected, the impact of each characteristic could have a dynamic effect on power consumption (*e.g.*, cache size and rate of instructions implemented). Therefore, an efficient method of search is needed to optimize the parameter set for a particular objective.

Table 5.1. Characteristic values for each parameter.

Name	Values
S_{IL1}: Instruction Cache L1 number of sets	S_{IL1} {128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K}
S_{DL1}: Data Cache L1 number of sets	S_{DL1} {128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K}
S_{UL2}: Unified Cache L2 number of sets	S_{UL2} {1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K}
B_{IL1}: Instruction Cache L1 block size	B_{IL1} {4, 8, 16, 32}
B_{DL1}: Data Cache L1 block size	B_{DL1} {4, 8, 16, 32}
B_{UL2}: Unified Cache L2 block size	B_{UL2} {8, 16, 32, 64}
A_{IL1}: Instruction Cache L1 associativity	A_{IL1} {1, 2, 4, 8}
A_{DL1}: Data Cache L1 associativity	A_{DL1} {1, 2, 4, 8}
A_{UL2}: Unified Cache L2 associativity	A_{UL2} {1, 2, 4, 8, 16}
BW_{L1L2}: Bus Width size from L1 to L2	BW_{L1L2} {32, 64, 128, 256}
N_{ALU}: number of integer ALU(s)	N_{ALU} {1, 2, 3, 4}
N_{MULT}: number of integer MULT(s)	N_{MULT} {1, 2, 3, 4}
FP_{ALU}: number of floating point ALU(s)	FP_{ALU} {1, 2, 3, 4}
FP_{MULT}: number of floating point MULTS(s)	FP_{MULT} {1, 2, 3, 4}
Q_{SIZE}: size of instruction queue	Q_{SIZE} {4, 8, 16, 32}
I_{RATE}: rate of instructions implemented	I_{RATE} {4, 8, 16, 32}

In conjunction with the variable parameters, there is a set of fixed parameters. Each fixed parameter is listed in a configuration file, as shown in Figure 5.4, and retrieved during the simulation process.

```
-max:inst           0 # maximum number of inst's to execute
-fastfwd            0 # number of insts skipped before timing starts
-fetch:mplat        3 # extra branch mis-prediction latency
-fetch:speed        1 # speed of front-end of machine relative to
                      execution core
-bpred              bimod # branch predictor type
                      {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod        2048 # bimodal predictor config (<table size>)
-bpred:2lev         1 1024 8 0 # 2-level predictor config (<l1size>
                      <l2size> <hist_size> <xor>)
-bpred:comb         1024 # combining predictor config
                      (<meta_table_size>)
-bpred:ras          8 # return address stack size (0 for no return
                      stack)
-bpred:btb          512 4 # BTB config (<num_sets> <associativity>)
-decode:width       4 # instruction decode B/W (insts/cycle)
-issue:inorder      false # run pipeline with in-order issue
-issue:wrongpath    true # issue instructions down wrong execution
                      paths
-commit:width       4 # instruction commit B/W (insts/cycle)
-ruu:size           16 # register update unit (RUU) size
-lsq:size           8 # load/store queue (LSQ) size
-cache:dl1lat       1 # l1 data cache hit latency (in cycles)
-cache:dl2lat       6 # l2 data cache hit latency (in cycles)
-cache:il1lat       1 # l1 instruction cache hit latency (in cycles)
-cache:il2          dl2 # l2 instruction cache config, i.e.,
                      {<config>|dl2|none}
-cache:il2lat       6 # l2 instruction cache hit latency (in cycles)
-cache:flush        false # flush caches on system calls
-cache:icompress    false # convert 64-bit inst addresses to 32-bit
                      inst equivalents
-mem:lat            18 2 # memory access latency (<first_chunk>
                      <inter_chunk>)
-tlb:itlb           itlb:16:4096:4:1 # instruction TLB config, i.e.,
                      {<config>|none}
-tlb:dtlb           dtlb:32:4096:4:1 # data TLB config, i.e.,
                      {<config>|none}
-tlb:lat            30 # inst/data TLB miss latency (in cycles)
```

Figure 5.4. Fixed parameter values during GA search.

Based on these parameterized characteristics and a particular embedded application, the search for an "ideal" configuration of an embedded processor is NP-complete [91].

5.1.1 Single-Objective Fitness Function

The single-objective fitness assessment is based on an estimate of power consumed for each configuration. Power models of common structures present in the embedded processor are used to estimate power consumed. Maximum power is calculated for the entire processor by summing power consumed in all datapaths, when all logic is used. Three conditional clocking schemes are used to estimate average power consumed. The conditional clocking (CC) scheme turns off devices that are not used in a particular cycle. CC1 considers a circuit to be 100% on if it is accessed and 0% if not. CC2 scales the power linearly with usage of the circuit. CC3 assumes linearly scaled power depending on usage when accessed, and a minimum of 10% of base power when the structure is not accessed. In the fitness function, the estimate from the third clocking scheme (CC3) is used. The fitness function is trained by the GA to optimize for the maximum. Since minimum power is desired, the inverse of the power consumed is the determining function for the fitness of a configuration. After the GA has reached its termination criteria, the best fitness solution for each generation is copied to an external file and converted back to actual power estimates.

5.1.2 Encoding and Decoding Algorithm

A look-up table is used for encoding and decoding parameter values. In GALOPPS, a one-dimensional array of possible values (codes) for each parameter is defined as 0, 1, 2, 3, *etc.*. The size of the one-dimensional array is the field size set for each parameter. For example, field_sizes[i], where $0 \leq i <$ number of fields, is equal to the number of different codes possible in the ith field (*i.e.*, codes will vary from 0 to field_sizes[i] – 1). These codes are represented in binary format (*i.e.*, 0000, 0001, 0010, ..., n), as shown in Figure 5.5. The chromosome length is 50 bits for the 16 fields. The fields only contain legal values for reproduction operation and selection.

SIL1[codes] ∈ {128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K}

field_size[i] ∈ {0,1,2,3,4,5,6,7,8}

binary_field[n] ∈ {0000,0001,0010,0011,0100,0101,0110,0111,1000}

Figure 5.5. Translation of chromosome to integer array.

5.1.3 Reproduction Operators

The two most common genetic operators are mutation and crossover. Mutation operates on a single chromosome to modify its characteristics through random manipulation of its genes. Crossover operates on two chromosomes, combining their genetic material to produce offspring (child) chromosomes. Mutation is good for global searches and crossover is good for local searches.

Mutation is done on a per-field basis, in the same way as for a binary representation where a bit, is a field. When a field is to be mutated, its value is changed uniformly at random to a different legal value. For our experiments the mutation rate was set to 0.014 per field (1.4%). The number of fields helped to determine the mutation rate. The goal was to achieve a 20% change in the population. Equation 5.1 illustrates how the mutation rate was determined:

$$P_m = 1-(\alpha^{(1/\mathit{fields})}) \tag{5.1}$$

$$P_m = 1-(0.8^{(1/16)})$$

$$P_m = 0.01378$$

where α is the percentage of chromosomes not mutated and *fields* is the number of non-binary fields per chromosome.

One point crossover for non-binary representation is performed only at the boundaries between fields. This restriction prevents generation of any illegal codes and preserves the fields as the basic elements of building blocks. The crossover probability controls how often the crossover operator is applied. The higher the crossover rate, the more quickly *new* candidate solutions are introduced into the population. If the crossover probability is too high, highly fit individuals are discarded faster than selection can produce improvements. On the other hand, if the crossover rate is too low, the search might stagnate for lack of exploration. For our experiments we chose the crossover rate to be 30%.

5.1.4 Selection Criteria

SUS, introduced by Baker in 1987, is a random selection technique with zero bias and minimum spread [76]. It is visualized as a pie chart, with lines of equal distance from each other, where *n* sections is the number of individuals in the population. The spinning wheel is placed on top of a pie chart that represents the percentage fitness value of each solution. The wheel is spun and the winners or selected individuals are those that have a line or pointer that stop in their space. In this process, an individual's selection probability is based solely on the initial spin and the magnitude of its expected value.

In order to increase the selection probability of "less fit" individuals, their fitness can be scaled within GALOPPS. Scaling of fitness values is implemented by assessing a scale factor. For example, if the scale factor is 1.5, and a set of fitness values is {0.5, 0.2, 0.1, 0.2}, with a mean of 0.25, the mean is subtracted from the fitness value. Then each new fitness value is divided by 2, which is derived from the ratio of the mean to the scaling factor, then added to the new fitness value for a final fitness {0.375, 0.225, 0.175, 0.225}. Fitness scaling offers a way to alleviate premature convergence.

5.2 Evolutionary Search Criteria

GAs can be applied effectively to configurable embedded processors in an effort to search for a desired optimum. In order to gain a better understanding of how the GA can search the design space effectively, a series of experiments were performed to validate the search criteria (*i.e.*, population size, number of trials, diversity of population) for the GA. The experiments used estimated values with a non-specified targeted workload for power consumption per configuration, using the design space of the 16 configurable parameters. A single-point crossover rate of 30% and mutation rate of 1.4% were used for the reproduction operators. The selection scheme for new chromosomes was SUS using linear scaling of fitness values. The experiments were performed for various population sizes, initial random populations and stopping criteria. The first experiment tested population sizes varying, from 50 to 100 with the stopping criterion of 50 generations. The second experiment tested various initial random populations and their final converging set of solutions. The third experiment tested number of trials (evaluations) versus population size for convergence of solution. This section illustrates the tradeoffs for an effective evolutionary search of an optimal configurable processor design.

5.2.1 Population Size

Population size can have an effect on testing time and the converging set of solutions. An experiment was conducted to determine whether a smaller population size can achieve similar results to a larger population size. The population was incremented by 10 from 50 to 100 to analyze the change in power consumption over a given population size. A population size of 70 yielded the best result with a minimal power consumption of 39.97 watts (Figure 5.6). A summary of population size effects and a list of the best configurations are provided in Table 5.2 and Table 5.3.

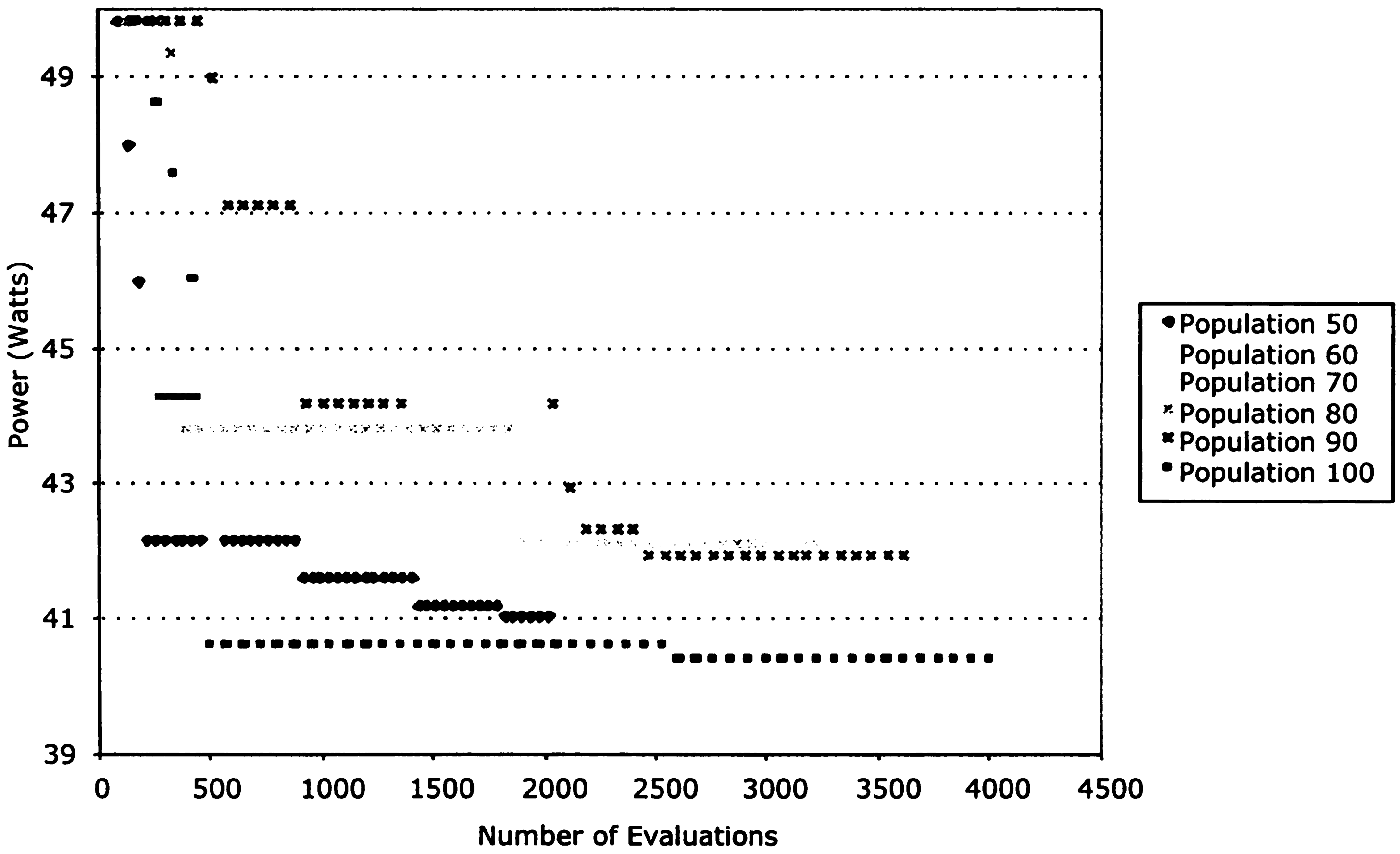

Figure 5.6. Experimentation results from varied population sizes.

The population of 50 had a faster converging rate and also terminated the GA process before other population sizes. If the termination criteria or number of generations were extended, then the population size of 50 may have a higher probability of finding a better solution.

Table 5.2. Analysis of population experiment.

Size of Population	50	60	70	80	90	100
Number of Trials	2017	2384	2801	3233	3609	3999
% Change in Power	21.31	21.58	24.60	18.30	18.76	23.21

Table 5.2 shows a comparison among population sizes. The range of power consumed from the initial configuration to the best final configuration varied from 18% to 25%. The population size of 70 had a greater diversity of fitness from its original configuration to its best configuration. If the number of trials were expanded for the population of 50, the diversity would also be greater. Also, from Table 5.2, it is noted that a population of 100 took twice as many trials as a population of 50. The third experiment will address the issue of convergence and number of trials needed in a GA run.

Table 5.3 shows the final configurations for each population size. From this table, a relationship can be established for some of the parameters, such as commonalities or inconsistencies.

Table 5.3. "Best" configuration of a GA run for each population size.

Parameter	pop_50	Pop_60	pop_70	pop_80	pop_90	pop_100
S_{DL1}	128	128	128	128	128	128
B_{DL1}	16	8	8	8	16	8
A_{DL1}	2	1	1	2	1	1
S_{IL1}	128	128	256	128	256	128
B_{IL1}	16	32	8	64	16	32
A_{IL1}	2	1	1	1	1	1
S_{UL2}	1024	4096	1024	2048	2048	1024
B_{UL2}	8	32	8	8	16	8
A_{UL2}	1	1	1	1	4	4
BW_{L1L2}	64	32	256	64	32	32
N_{ALU}	1	1	1	1	1	1
N_{MULT}	2	2	4	3	4	4
FP_{ALU}	1	1	1	1	1	1
FP_{MULT}	2	2	3	4	2	4
Q_{SIZE}	32	8	32	32	32	4
I_{RATE}	4	4	4	4	4	4
Power (watts)	41.0553	40.9634	39.9699	42.099	41.9374	40.4216

In the different GA runs, with a termination criterion of 50 generations, the final configurations were not all the same. The configuration with the lowest power consumption was created from a population size of 70. The optimal configuration for maximum power after several generations of different population sizes converged to smaller numbers for specified parameters. The important reason for running this test was to determine which population size would generate the best solution within a short period of time. However, with only one random initial population, the search exploration space is limited, thus not giving a true representation of the solution search space.

5.2.2 Random Initial Populations

In GALOPPS, a random seed can be set to initiate a certain population in the design space. Each randomly seeded population can represent different areas of the search space. By using different random seeds, a confidence level of search quality can be satisfied. The following experiment launches five random initially seeded populations with varying sizes (50, 60, 70, 80, 90, 100). Figure 5.7 illustrates the best fitness for each random seeded population.

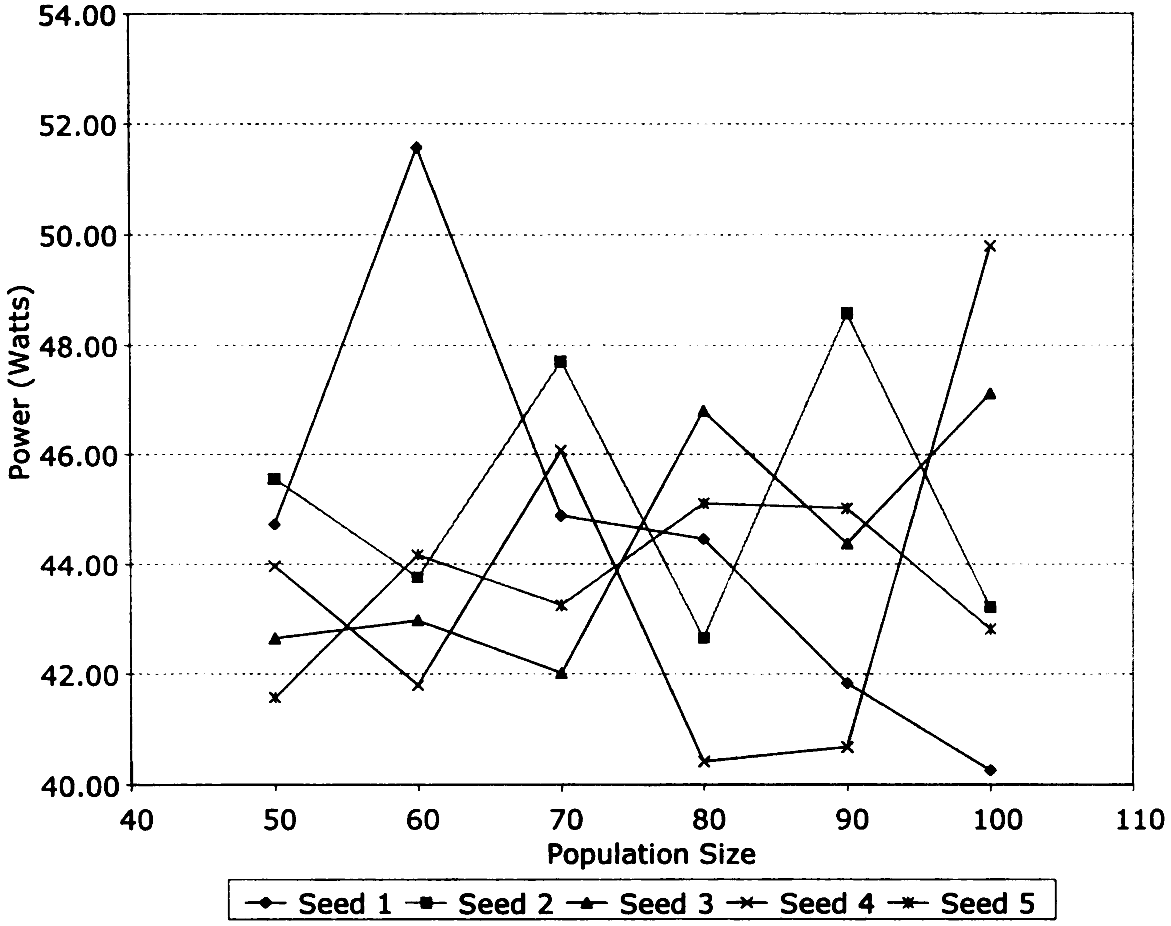

Figure 5.7. The best fitness for each random seeded population.

One misconception some researchers have is that one run (*i.e.*, GA exploration from start population to end population) is good enough to establish a reasonable solution. In this experiment, after running the GA for each randomly seeded population, each population displayed varying fitness results. If only a single GA run was explored, the search space could be limited to a particular area.

On average, the population size of 50 had the lowest fitness for power, as shown in Table 5.4.

Table 5.4. Fitness of each random seeded population.

Population	**50**	**60**	**70**	**80**	**90**	**100**
1	44.73	51.57	44.88	44.45	41.83	40.26
2	45.56	43.76	47.70	42.64	48.57	43.21
3	42.64	42.98	42.02	46.80	44.37	47.12
4	43.96	41.80	46.08	40.41	40.67	49.79
5	41.58	44.17	43.25	45.10	45.01	42.82
Average	43.69	44.85	44.79	43.88	44.09	44.64
% Difference	0.00%	2.66%	2.50%	0.43%	0.91%	2.16%

The percent difference for the best average fitness was within a 3% margin. Initial randomly seeded populations randomly select starting populations to generate a formal structure of the design search space. This is similar to selecting a point from a two-dimensional graph in the design search space. The point represents a unique solution. By randomly sampling five or more initial populations, there is a probability of selecting a unique solution from one or all quadrants. A closeness of optimum can be estimated using randomly seeded populations.

5.2.3 Termination Criteria

The GA is used to find the "best" or "ideal" solution, which if implemented correctly, will find a "good" or satisfactory solution in a reasonable amount of time. Assessing the convergence of the population is another technique used to determine whether a GA has reached a stopping point. For example, the third experiment examines the size of a population versus the number of trials to determine whether a smaller population would converge faster than a larger population when optimizing for power consumption of a configurable processor.

In the previous experiment, a population size of 50 had a lower "best" fitness average, which serves as the smaller population size. To choose the larger population size, a series of GA runs were done for population sizes of 100, 200, and 400. With a population size of 400 and a stopping criterion of 50 generations, there were a total of 9243 evaluations. The minimum amount of power computed was 44.75 watts, for 5 initial random populations in 13 hours. Since computation time is an important factor, the large population size was set at 400. In order to compare the two population sizes, 50 and 400, the termination criterion for the population size of 50 was set to 400 generations, which would generate at least 9000 trials. The best fitness for each generation is shown in Figure 5.8.

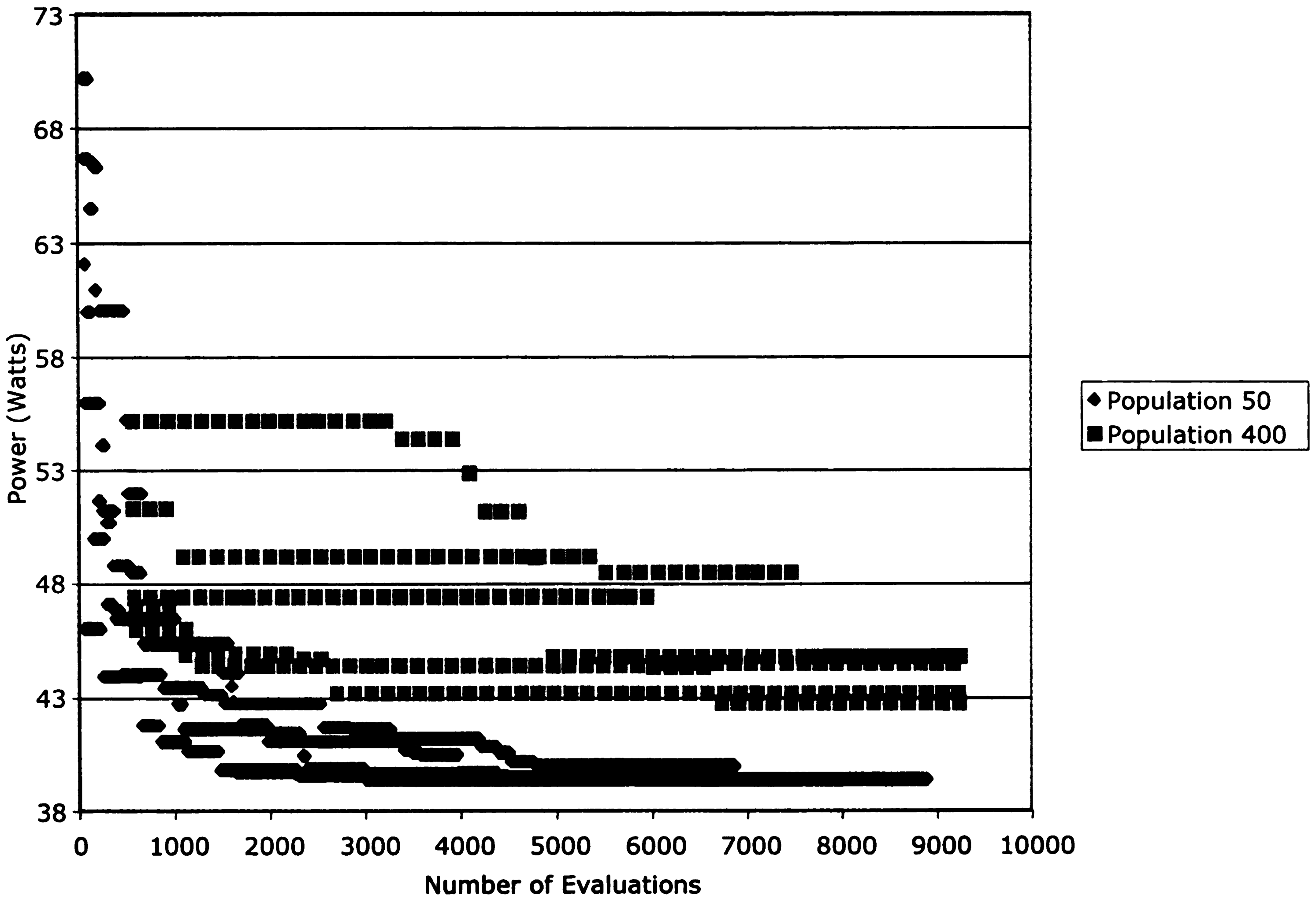

Figure 5.8. Fitness evaluations for an extended run of generations.

A population size of 50 generated convergence with half as many trials as the population of 400, as shown in Tables 5.5 and 5.6. In Table 5.5, each random seed population is listed with its best fitness for its initial population, converging value for power consumption, percent difference, and number of trials. The number of trials represents the evaluations needed to attain a calculated power consumed. The converging value for all randomly seeded populations was 39.37 watts. The configurations, however, continued to fluctuate until the end of the maximum number of generations. Also noted from Table 5.5 is the percent difference, where the highest percentage generated a faster convergence.

Table 5.5. Population size of 50 for 400 generations.

Random Seed	1	2	3	4	5	Average
Reference Power	70.15	55.92	66.63	62.04	45.97	60.14
Ending Power	39.37	39.37	39.37	39.37	39.37	39.37
% Difference	78.17	42.03	69.23	57.56	16.76	52.75
Number of Trials	3010	5920	4402	6889	4513	4947

In Table 5.6, the number of trials is the maximum number of evaluations per GA run. None of the ending power consumption values were similar, and the percent difference between reference power and ending power were smaller in comparison to the population size of 50. For the larger population to converge, the maximum number of generations would need to be extended, thus increasing the amount of computation time. For the larger population to reach the minimum convergence of a population of 50, more trials are also needed. In addition, changing the mutation rate can generate more diversity in the population, thus increasing the rate of convergence.

Table 5.6. Population size of 400 for 50 generations.

Random Seed	**1**	**2**	**3**	**4**	**5**	**Average**
Reference Power	46.97	55.14	51.27	47.38	45.97	49.35
Ending Power	43.17	44.77	44.76	44.48	42.75	43.98
% Difference	8.81	23.16	14.55	6.54	7.53	12.19
Number of Trials	9222	9188	9243	9161	9234	9210

These experiments showed that with a smaller population size, an estimated optimum can be reached at a faster rate. Also, random initial populations can inform the user of probable convergence. The experiments conducted serve as the basis for understanding the genetic search technique, and will be used as a foundation for further evaluation and testing of a configurable embedded processor.

CHAPTER 6

Multi-Objective Search Optimization

The optimization process presented in Chapter 5 attempts to optimize for a single objective, the estimated power consumption of an embedded processor, for a particular application. However, practical embedded processor design involves the simultaneous optimization of a number of objectives such as power, speed, area, cost, *etc.*

Within the framework of a GA, multiple parameters are integrated into the optimization process, where all objectives are simultaneously optimized. In other words, no single objective is concentrated on at the expense of others. The simultaneous optimization of all objectives will produce a single solution. One technique for implementing multiple objective optimization is the use of a weighted fitness function [79,80] given as

$$\text{Total\_Fitness} = \alpha^*f(x_1) + \beta^*f(x_2) + \ldots + \zeta^*f(x_n). \qquad (6.1)$$

Equation 6.1 demonstrates that a typical implementation of a weighted function is the sum of all the individual fitness values for each of the functions, from 1 to n different objectives. The variables α, β, and so on, are used to weight the individual contribution of each parameter to the overall fitness. The use of a weighted fitness function combines the discrete fitness values for each function into a scalar fitness value. The scalar fitness value is used to assess the overall quality of the solution.

One of the main problems with such a technique is the assignment of the weighted objectives. These weighted objectives significantly affect the performance of the system in determining the globally optimal result [81]. The weights effectively place a priority on the optimization of a particular objective in relation to the optimization of other objectives. Thus, assigning the correct priority to the multiple objective functions is required to enable the optimization process to produce a globally searched optimum solution.

In practical engineering problems, prioritization of multiple objectives is a complex and difficult process. Such problems are often characterized by a number of competing objectives where improvements in one objective overshadow the cost of degrading the other objectives. Furthermore, the tradeoffs between competing objectives are often non-linear. For example, consider the bi-objective problem of designing an embedded processor for minimum power and maximum performance. The two objectives are in competition with each other, as many low power design techniques use smaller cache sizes to decrease power. In a practical design process, the embedded processor may be targeted for a certain rate of speed. An increase in cache size may result in a large increase in cost and decrease in fitness, whereas small increases will have little effect on cost and overall fitness. Therefore, the effect on performance is dependent upon the actual fitness value. A single weight to assign a specific priority to the performance objective would not accurately reflect the total fitness for all solutions.

If the non-linear nature of the objectives can be overcome, there is still the problem of assigning a relative priority to each function for all cases, (*e.g.*, is performance more important than power, and if so, how much more?). The example illustrates the difficulty of combining distinct parameters into a single fitness evaluation to generate a single globally optimal solution. An alternative to a weighted fitness function is to use the GA to explore the solution space and present a range of alternative non-dominated solutions (NDS) that are each optimal for a single function [81,82]. This removes the need to prioritize parameters during the optimization process. The experienced designer can then analyze the alternative solutions to select the solution that best satisfies the specified requirements.

During the optimization process, the GA evaluates many alternative solutions, which can be useful information when presenting the chosen solution. This information can be used to illustrate tradeoffs between different parameters in the optimization problem. The tradeoff between competing objectives is usually presented as a Pareto-optimal front also known as Pareto-surface, Pareto-points, and Pareto-optimal set [63]. The use of Pareto-points collected throughout the GA optimization process has been previously shown to be beneficial to the low power design process [58,83,84].

6.1 Pareto-Optimal Front

A Pareto-optimal front is a set of NDS, where each point on the Pareto-surface has no better values for those objectives. Figure 6.1 shows an example of a Pareto-optimal front for a two-dimensional problem.

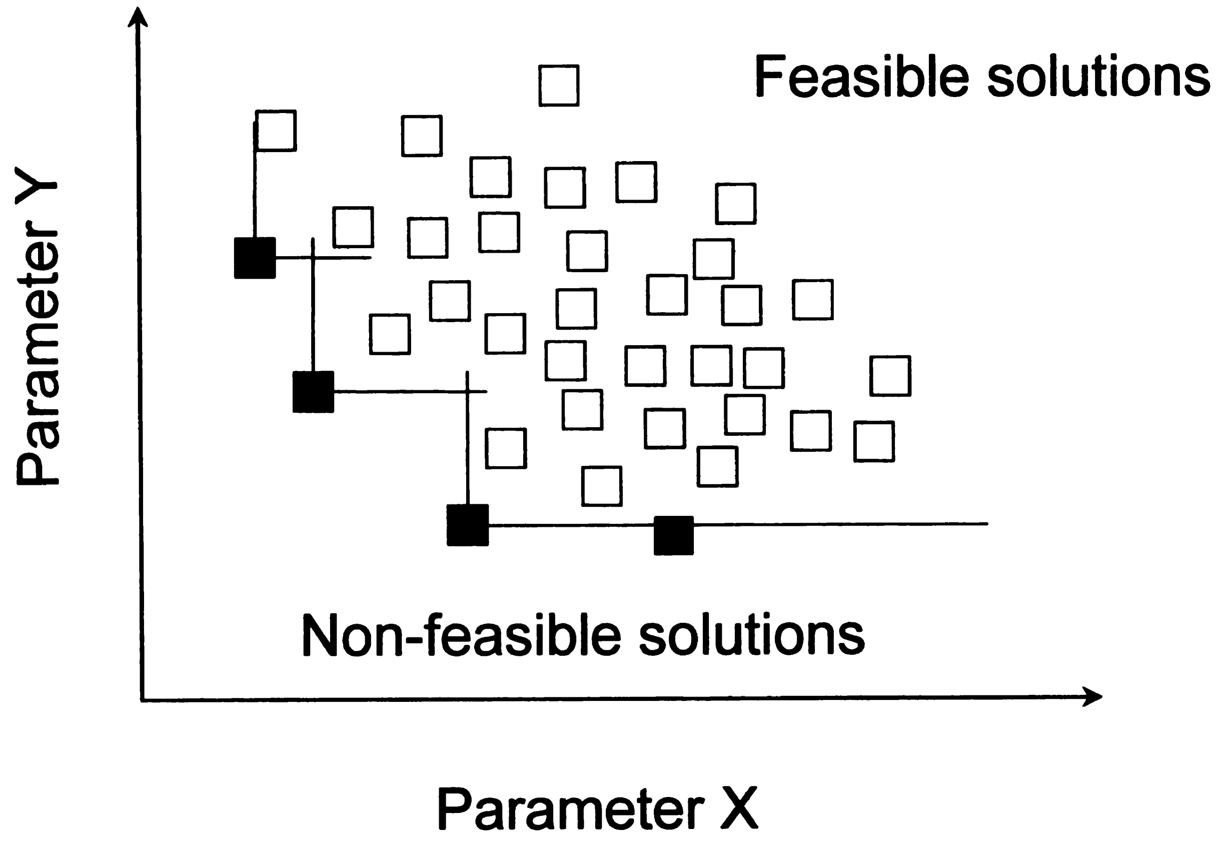

Figure 6.1. Pareto-optimal front for a two-dimensional problem.

The chart illustrates a typical engineering design example where two competing objectives (Parameter X and Y) must be minimized. Each solution is illustrated on the chart as a square. The NDS points, shaded squares, are known as Pareto-optimal. The Pareto-optimal front is a curve joining the set of Pareto-points. These points mark the boundary between the range of feasible and non-feasible solutions, illustrating the trade-off between each function.

The set of NDS does not present an obvious single optimum solution. Selection of the best solution is left to the design architect, dependent upon the priority placed on each objective. One of the main advantages of the Pareto-chart is that it shows the effect of varying the importance of a function, as opposed to presenting the architect with a single point solution. For example, the chart may show that a small variation in X, previously set as a constraint, may allow such a large reduction in Y as to make the design feasible. The Pareto-chart also allows the designer to use expert knowledge of the problem to select an optimum solution.

The range of alternative solutions may be more useful than a single point for the next stage of the design and implementation process. The presentation of design alternatives within a CAD simulation tool is regarded as essential by most designers [85]. The presentation of a set of optimal solutions enables the designer to gain a greater understanding of the low power solution space and the power characteristics of the problem to be optimized.

6.2 Pareto-Optimal Front Generation in GALOPPS

A Pareto-point is defined as that which has no lower value in both the X and Y axes for performance and power. The identification of Pareto-points is split into a two stage process, where all designs with the lowest Y value for every generated X value and vice versa are first identified. A weighted value is assigned and the data for X and Y are identified. Then, the next stage steps through the set of generated points, and sorts for NDS in ascending X values. This process is illustrated in Figure 6.2.

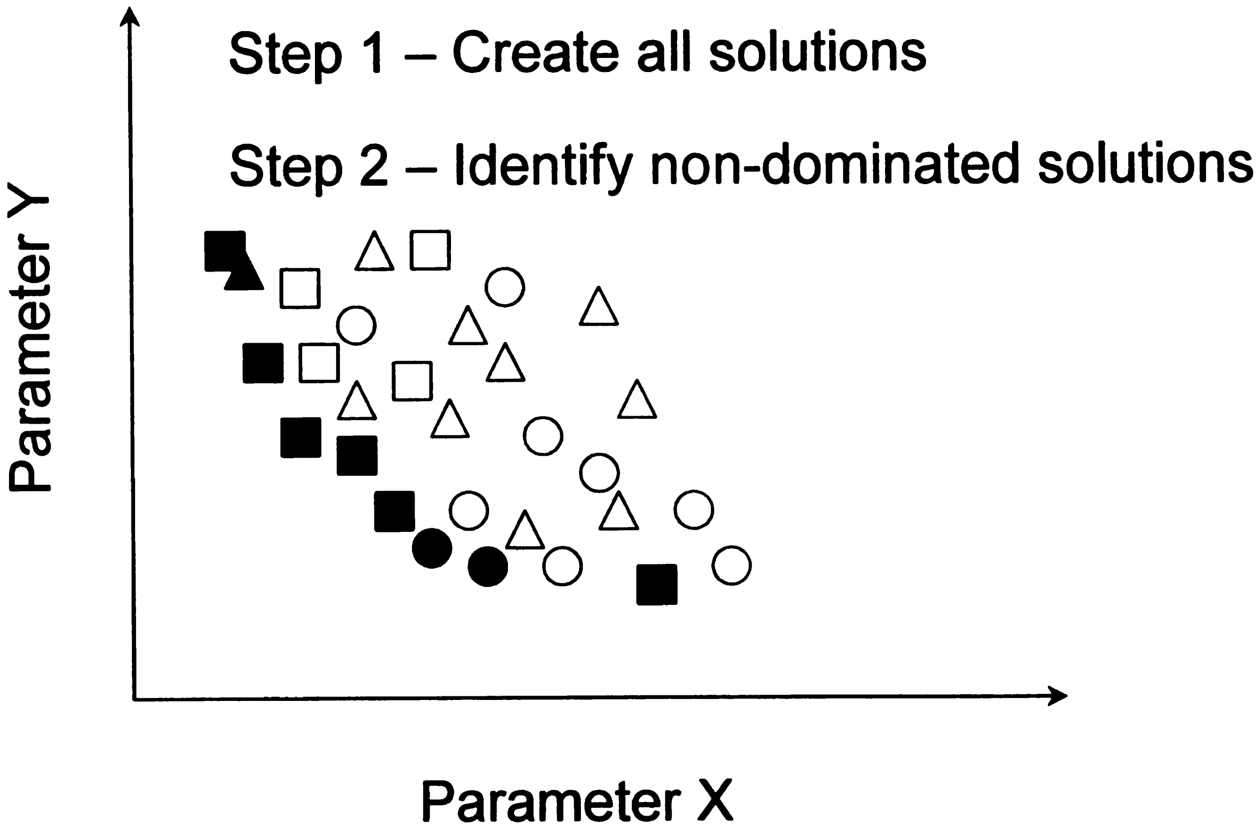
Step 1 – Create all solutions
Step 2 – Identify non-dominated solutions
Parameter Y
Parameter X

Figure 6.2. Identification of Pareto-points in GALOPPS.

Executing both stages produces a set of NDS, the power-performance Pareto-optimal front. After each generation is created the power and performance of each design is analyzed to create a list of points (*i.e.*, power and performance estimation explored during the simulation process). After the GA has determined the lowest power solution and lowest performance (CPI) solution, stage 2 is executed to determine which of these points are Pareto-optimal.

6.3 Power and Performance Trade-off for Benchmark Designs

This section illustrates the power and performance trade-offs, through the use of Pareto-optimal analysis, for benchmark designs. The Pareto-charts were generated using the techniques described in Section 6.2. Each Pareto-chart is presented in a separate section for that design. In each chart a solid line denotes the Pareto-optimal front. The Pareto-surface is presented as a straight line instead of a curve joining the Pareto-points. The solutions represent discrete points in the search space, where if the point were joined by a curve, it may imply that there is a range of solutions between two points when no feasible solution exist between those points.

Each chart illustrates the unique points in the solution space examined while searching for an optimum solution for that design. Each 'X' denotes a unique power-performance point for a design. It should be noted that each power-performance point does not necessarily correspond to a single design configuration. A number of designs can have the same power-performance value, as illustrated in Figure 6.3. Therefore, the number of ‘X’ points in the graph may not be representative of the total number of designs analyzed throughout the search process.

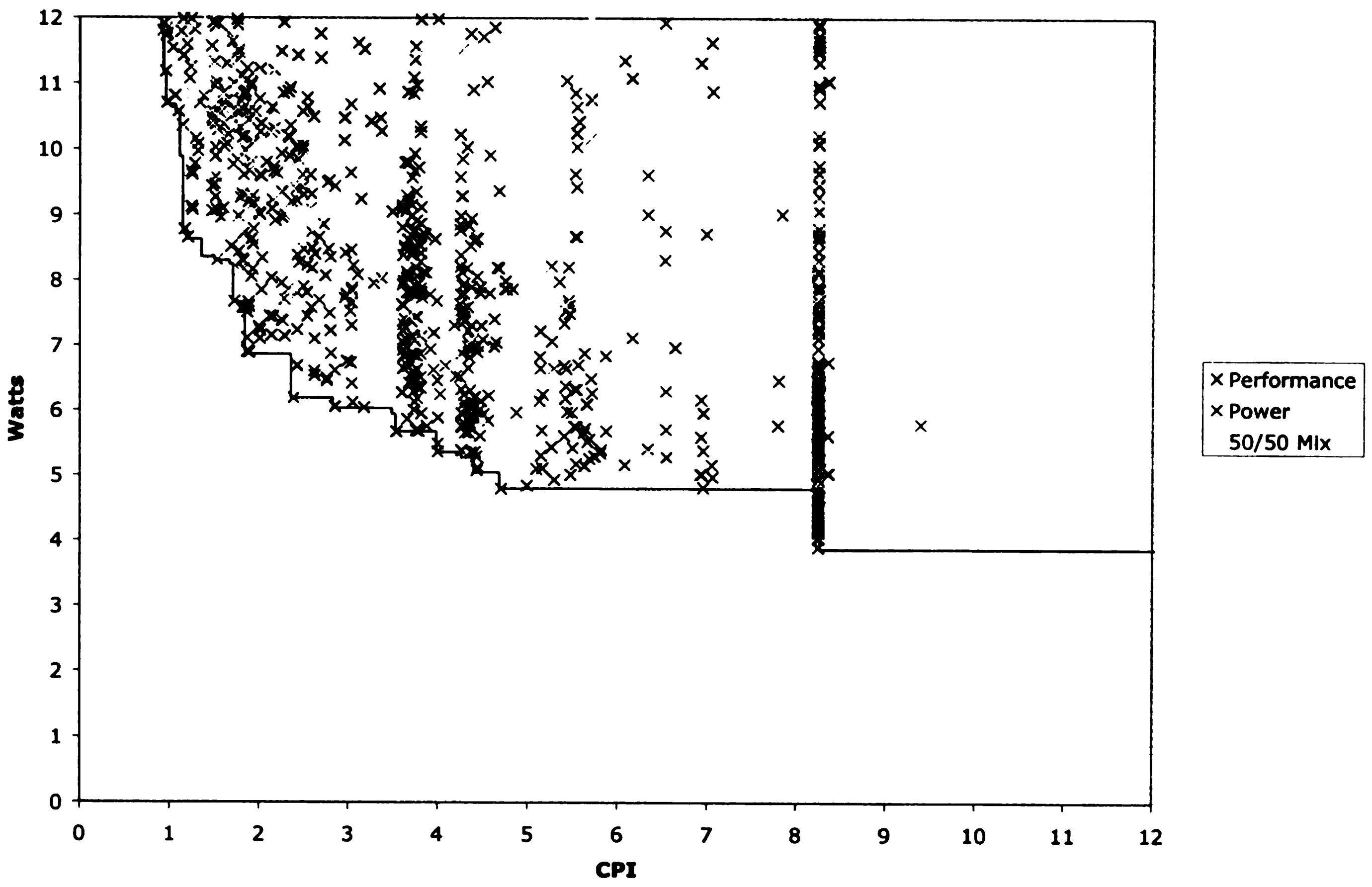

Figure 6.3 Power-Performance tradeoff using Pareto-chart.

6.4 Test Configurations and Benchmarks

In this section, a comparison of published maximum power numbers for three processors is presented. These three processors are from MIPS, Wattch, and a study done at the University of Pennsylvania [10,86]. The configuration for the MIPS processor is taken from the R10000 specifications [10]. The Wattch configuration is a baseline specification for the SimpleScalar processor [10]. The study done at the University of Pennsylvania demonstrated different tradeoffs between configurations for improved IPC [86]. The baseline configuration is used to represent academic standards.

Table 6.1. Test configurations for power-performance tradeoffs.

	R10000	UPENN	Wattch
fetch:ifqsize	8	4	4
fetch:mplat	1	3	3
fetch:speed	1	1	1
Bpred	Bimod	bimod	Bimod
bpred:bimod	2048	128	2048
bpred:2lev	1:1024:8:0	2:1024:6:0	1:1024:8:0
bpred:comb	1024	1024	1024
bpred:ras	8	8	8
bpred:btb	512 4	256 4	512 4
decode:width	4	4	4
issue:width	4	4	4
issue:inorder	FALSE	TRUE	FALSE
commit:width	4	4	4
Ruu:size	64	64	16
Lsq:size	16	16	8
cache:dl1	dl1:512:32:2:l	dl1:512:32:2:f	dl1:128:32:4:l
cache:dl1lat	1	1	1
cache:dl2	Ul2:16384:64:2:l	ul2:2048:64:4:l	ul2:1024:64:4:l
cache:dl2lat	8	1	6
cache:il1	il1:512:32:2:l	il1:512:32:2:f	il1:512:32:1:l
cache:il1lat	1	1	1
cache:il2	dl2	dl2	dl2
cache:il2lat	8	6	6
mem:lat	40 2	32 1	18 2
mem:width	16	8	8
Tlb:itlb	itlb:16:4096:4:l	itlb:1:4096:16:l	itlb:16:4096:4:l
Tlb:dtlb	dtlb:16:4096:4:l	dtlb:1:4096:32:l	dtlb:32:4096:4:l
Tlb:lat	30	20	30
Res:ialu	2	2	4
Res:imult	1	1	1
Res:memport	2	1	2
Res:fpalu	2	1	4
Res:fpmult	1	1	1

The experiments used estimated values with a targeted workload (*test-math.c* and *126.gcc*) for power consumption and cycles per instruction (CPI) per configuration. Three objective fitness functions, (power, performance, 50/50 weighted mix) were used to guide the GA during the search process. Five initial randomly seeded populations were submitted to the GA for diverse exploration of the search space. The population size for each run was 50, with termination set at a maximum of 50 generations. A single-point crossover rate of 30%, and mutation rate of 1.4% was used for reproduction operators. The selection scheme for new chromosomes was SUS using linear scaling of fitness values.

To illustrate the effectiveness of the GA and the algorithms presented, the performance of the prototype version of the GA embedded configuration simulation tool is demonstrated with a set of benchmark example designs. The two benchmarks chosen for this application are *test-math.c* and *126.gcc* (*cc1*), where *cc1* is part of an industry standard benchmark suite. The *test-math.c* program illustrates the use of trigonometric functions and tests these functions to verify validity of calculations. The *cc1* benchmark is a GNU C compiler. As discussed previously in Chapter 4, the C program is translated and mapped to a virtual instruction set architecture, then applied to a specified configuration of parameters to simulate the actual processor. The benchmark designs presented are used both to illustrate the effectiveness of the simulation tool and to investigate improvements and additions to the overall toolset. Tables 6.2 and 6.3 lists the results obtained for the example processor configurations and the configurations derived using multi-objective GA optimization.

Table 6.2. Comparison results from Math Benchmark.

Percent Difference of Initial GA config.	GA Optimization	R10000	PENN	Wattch	Multi-Object GA
Power	-125	16.32	42.07	37.28	21.99
CPI	12.56	-28.69	-100.23	-43.14	1.72
Total	-112.44	-12.36	-58.16	-5.85	23.72

Table 6.3. Comparison results from GNU C Compiler Benchmark.

Percent Difference of Initial GA config.	GA Optimization	R10000	PENN	Wattch	Multi-Object GA
Power	34.71	14.19	43.94	30.09	49.17
CPI	9.99	-26.69	-112.16	-14.67	0.42
Total	44.69	-12.50	-68.23	15.42	49.59

Tables 6.2 and 6.3 shows the results obtained from the test configurations and the configurations derived in our methodology. Since performance is the leading objective for these examples, the original configuration is taken from the best solution in the first run of the GA, for performance optimization. The results are expressed as a percentage of the original designs' power consumption and CPI performance. The percentage illustrates the improvement in power consumption and performance derived through the multi-objective GA optimization tool. The following sections discuss the results obtained with each of the benchmark designs.

6.4.1 Math Benchmark

The *test-math.c* benchmark illustrates the use of trigonometric functions and tests these functions to verify validity of calculations from the simulated processor. The industry example, MIPS R10K simulated processor showed an improvement of 16.32% in power consumption, but a 28.69% decline in CPI performance in comparison to the original configuration. In comparison with the other specified processor configurations, the R10K trades power consumption for higher performance. The Wattch example showed an improvement of 37.28% in power consumption, but a 43.14% decline in CPI performance. The Wattch example considers both power and performance for an overall better processor. The configuration used in a study from the University of Pennsylvania showed lower performance and more power consumption in comparison to the other examples.

The results from the GA runs, with a population size of 50, generated a processor configuration with a better CPI performance and, in a separate run, with less power consumption, as shown in Table 6.4 and 6.5. Table 6.4 lists the results from a GA run that optimized for CPI performance. There were four initial randomly seeded populations. The maximum CPI for the best configured processor in the initial run was 0.72. It took 1173 trials, 20 computational hours, to reach a minimum of 0.63 CPI. Random seeded population #4 had the largest percent difference from original best configuration to final best configuration, and the percent difference between final best configuration were within 3%.

Table 6.4. GA results for CPI performance optimization.

	Seed 1	Seed 2	Seed 3	Seed 4
Orig. CPI	0.72	0.67	0.66	0.72
End CPI	0.65	0.64	0.65	0.63
# Trials	1155	1147	1163	1173
% Diff Orig.	11.20%	4.89%	0.54%	14.36%
% Diff Best	1.83%	0.31%	2.90%	0.00%

Table 6.5 lists the results from a GA run that optimized for power consumption. There were five initial randomly seeded populations. The maximum power calculated for the best configured processor in the initial run was 6.89 watts. The minimum power consumed for a configured processor was 3.84 watts, with 69.87% difference from the original best configuration. In this GA run, there is 10% difference from the lowest to highest power consumed for a final processor configuration. This difference is somewhat higher than expected and, with more trials, the difference can possibly be reduced thus closely approaching an estimated optimal configuration for all seeded populations.

Table 6.5. GA results for power consumption optimization.

	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5
Orig. Power	6.89	4.82	6.49	6.28	4.80
End Power	4.25	3.90	3.82	3.84	3.84
# Trials	1140	1149	1143	1149	1155
% Diff Orig.	62.13%	23.57%	69.87%	63.40%	25.06%
% Diff Best	10.15%	2.12%	0.00%	0.67%	0.58%

However, for each GA run the other objective suffered. With performance being the main objective in the case of the math benchmark, the calculated power consumed showed a decline of 125% in comparison to the original best configuration. The multi-objective weighted Pareto-chart illustrated a better tradeoff of power-performance in comparison with other processor configurations, as shown in Table 6.2. The actual Pareto-chart for all of the combined GA runs is shown in Figure 6.4.

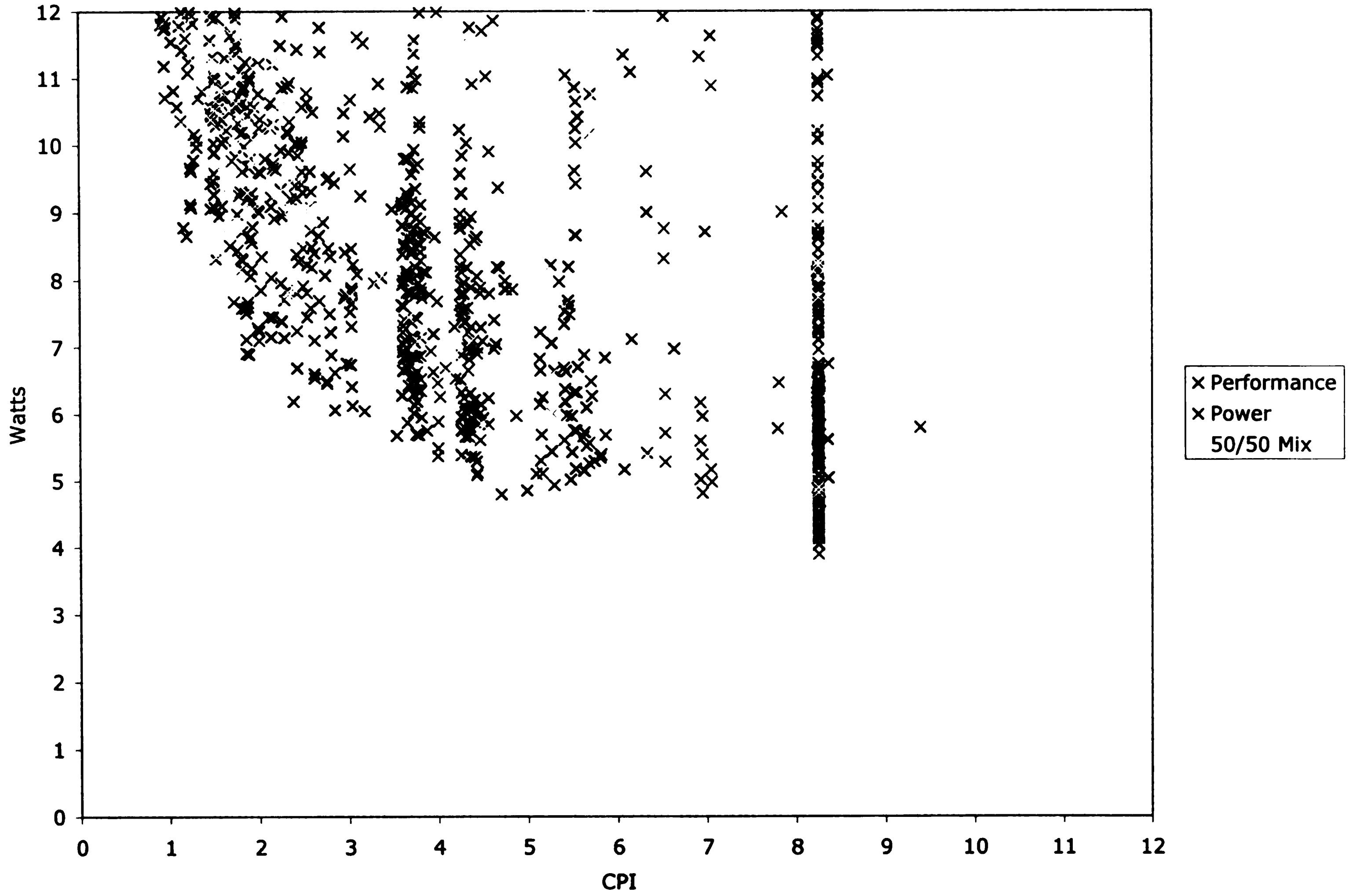

Figure 6.4. Pareto-chart of *test-math.c* benchmark.

6.4.2 GNU C Benchmark

The *126.gcc* benchmark illustrates the compilation of GNU C compiler. This benchmark belongs to a set of tests produced by the Standard Performance Evaluation Corporation (SPEC) [87]. These tests were developed to provide comparable measures of performance for compute-intensive workloads on different computer systems. The industry example, MIPS R10K processor showed an improvement of 14.19% in power consumption, but a 26.69% decline in CPI performance in comparison to the original configuration. In comparison with the other specified processor configurations, the R10K trades power consumption for higher performance. The Wattch example showed an improvement of 30.09% in power consumption, but a 14.67% decline in CPI performance. The Wattch example considers both power and performance for an overall better processor. Again, the configuration used in a study from the University of Pennsylvania showed lower performance and more power consumption in comparison to the other examples.

The results from the GA runs, with a population size of 50 generated a processor configuration with a better CPI performance and in a separate run with less power consumption, as shown in Table 6.6 and 6.7. Table 6.6 lists the results from a GA run that optimized for CPI performance. There were four initial randomly seeded populations. The maximum CPI for the best configured processor in the initial run was 0.76. It took 1121 trials, 20 computational hours, to reach a minimum of 0.68 CPI. Random seeded population #4 had the largest percent difference from original best configuration to final best configuration, and the percent difference between final best configuration were less than 0.5%.

Table 6.6. GA results for CPI performance optimization.

	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5
Orig. CPI	0.69	0.74	0.71	0.76	0.70
End CPI	0.68	0.68	0.68	0.68	0.68
Trials	1148	1153	1121	1141	1143
% Diff Orig.	1.06%	7.69%	3.36%	11.06%	3.03%
% Diff Best	0.00%	0.29%	0.00%	0.03%	0.00%

Table 6.7 lists the results from a GA run that optimized for power consumption. There were five initial randomly seeded populations. The maximum power calculated for the best configured processor in the initial run was 8.30 watts. The minimum power consumed for a configured processor was 3.82 watts, with 64.37% difference from the original best configuration. In this GA run, there is 7% difference from the lowest to highest power consumed for a final processor configuration. This difference is somewhat higher than expected and, with more trials, the difference can possibly be reduced thus closely approaching an estimated optimal configuration for all seeded populations.

Table 6.7. GA results for power consumption optimization.

	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5
Orig. Power	8.30	4.82	7.73	6.28	4.80
End Power	3.88	3.88	4.12	3.82	3.92
# Trials	1120	1155	1158	1143	1185
% Diff Orig.	113.97%	24.37%	87.68%	64.37%	22.41%
% Diff Best	1.52%	1.40%	7.28%	0.00%	2.61%

However, for each GA run the other objective suffered. With performance being the main objective in the case of the GNU C compiler benchmark, both were improved, with performance 10% and the calculated power consumed was 30% better than the original best configuration. Using the multi-objective search, we were able to select a configuration with improved power consumption of 49.17%, sacrificing performance with a gain of 1%. The total in power-performance savings was ~50% compared to single-objective optimization total of ~45%, as shown in Table 6.3. The actual Pareto-chart for all of the combined GA runs is shown in Figure 6.5.

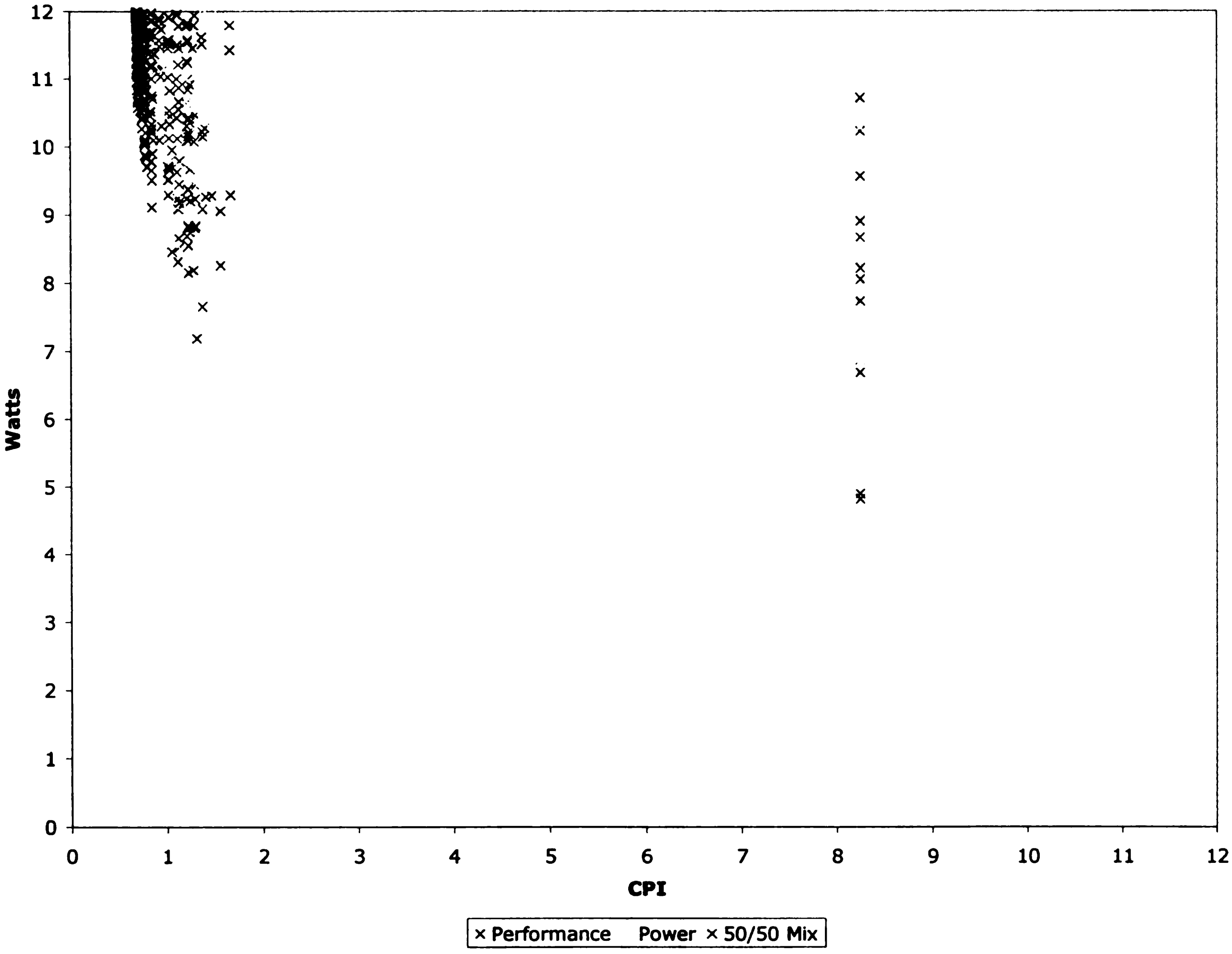

Figure 6.5. Pareto-chart for GNU C Compiler benchmark.

Optimization problems inherently involve optimizing objectives subject to various specifications and constraints. In a single objective problem, the goal is to find the "best" solution that maximizes or minimizes the objective, while in a multi-objective optimization problem the goal is to arrive at a set of Pareto-optimal designs. The useful feature of Pareto-optimal designs is that they are diverse, where they provide a wide choice for the decision-maker. Classical optimization methods are in general not efficient for multi-objective problems as they often lead to a single solution instead of a set of final solutions. Multiple runs of the same method cannot guarantee a different point on the Pareto-front each time and some methods cannot handle problems with multiple optimal solutions. Evolutionary methods maintain a set of solutions as a population during its course of search and thus can result naturally in a set of Pareto-optimal solutions in a single run. From the research presented, a widely differing set of Pareto-optimal solutions can be generated using weighted strategy within the evolutionary algorithm, therefore enlarging the population of designs to provide a wide choice for the designer.

6.5 Sensitivity Analysis of Parameters

Sensitivity analysis attempts to understand the uncertainties of various input parameters used in the optimization of power and performance. Sensitivity refers to a model's response to parameter changes. Pruning is the procedure where inputs are removed based on their contribution or sensitivity during the search. A sensitivity analysis method was used to determine the relative contribution of each parameter and then eliminated or replaced non-significant parameter values with the best value obtained from the search. In this case, the best value is the value at the end of a GA run for a particular parameter.

For example, each solution includes 16 changeable parameters in the configuration of an embedded processor, with a minimum of four options per parameter, we would be able to obtain $4^{16} \approx 8.15 \text{ X } 10^9$ configurations. If two parameters were taken away and made constant, the search space reduces to $2.68 \text{ X } 10^8$, thus the search is refined and more concentration can be placed on other parameters. Each parameter is evaluated for sensitivity by counting the number of mutations of the "best fit" individual in a given run, and least sensitive parameters are replaced by constant values.

The same setup of experiments was used in the new GA model. The experiment used estimated values with a targeted workload of *test-math.c* for power consumption and cycles per instruction (CPI) per configuration. Three objective fitness functions, (power, performance, 50/50 weighted mix) were used to guide the GA during the search process. Five initial random seeded populations were submitted to the GA for diverse exploration of search space. The population size for each run was 50, with termination set at a maximum of 50 generations. A single-point crossover rate of 30%, and mutation rate of 1.4% was used for reproduction operators. The selection scheme for new chromosomes was SUS using linear scaling of fitness values. From the sensitivity analysis, certain parameters that were determined to be less sensitive (*e.g.*, data cache size, integer and floating point ALUs) used final values from the original GA run in the new GA model, and were not included in the exploration of the search space.

For example, Figure 6.6 illustrates a normalized view of parameter sensitivity. Three different GA runs were plotted: performance, power, and a 50/50 weighted mix of performance-power. For performance, NMULT, the number of integer multiplier units hasthe lowest switching activity during the GA run. The next lowest parameters are NALU and FPALU. For the new model, the final values for those parameters are selected from the configuration with best performance and are used as constants.

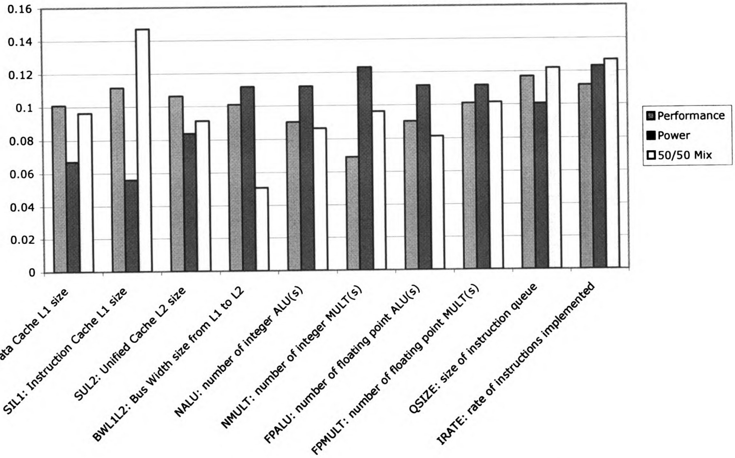

Figure 6.6. Parameter sensitivity of GA for performance, power, and weighted mix.

Once a new GA model is generated for each objective, the models are then tested and compared to the original run to see if there is any change in final fitness evaluations or variation in parameter sensitivity. From these experiments, the new model produced similar results to the original set of experiments. The next step is to further reduce the design space by reapplying the sensitivity analysis. The continuation and evaluation of the sensitivity algorithm is an interesting topic for further research and this idea is expanded on in Chapter 7.

CHAPTER 7
Conclusions and Scope of Future Work

This chapter details the main conclusions of the research presented in this thesis. Section 7.1 highlights the primary contributions of this work to the fields of portable embedded system architecture design. Section 7.2 gives an overview of the developed system. Section 7.3 discusses the results obtained in the context of power-performance design implications. Section 7.4 presents a summary of the conclusions derived from this work. Finally, Section 7.5 discusses future directions and developments for work initiated in this thesis.

7.1 Contributions

The primary, original contribution of this thesis is the development of a novel power-performance design tool based around a core GA search and optimization technique. The tool targets the implementation of portable embedded applications. The summary of power and performance estimation techniques at high levels of the design process, Chapter 2, was used to illustrate that targeting two objectives offers greater benefits than just a single objective. Thus, the developed tool targets power and performance reduction at the architectural configuration level.

System level optimization required the development of a non-standard chromosome representation for the GA. The chromosome does not use standard binary or alphabet chromosome representation, but incorporates the features of a configurable processor. The flexible nature of the chromosome ensures that it can represent a wide range of solutions of varying size and complexity, both initial designs and those generated throughout the search and exploration process. The ability of the GA to provide trade-off information to the design engineer was also presented, illustrating the advantages of using a GA for high-level design and architecture trade-off exploration.

7.2 System Implementation

The design tool was implemented in the C programming language, where the software was developed and tested using a Unix workstation running under the Solaris operating system. The design tool is comprised of the SimpleScalar performance simulator developed by Burger *et. al.* at the University of Wisconsin [21], Wattch power simulator developed at Princeton by Brooks *et. al.* [10], and GALOPPS developed at Michigan State University by Goodman *et. al* [75]. Additional files developed for simulation and GA evaluation include the application file, application input file, reading, stripping, and data sorting.

7.3 Discussion

The comparative analysis of original and optimized designs was performed with the use of high-level power and performance estimation strategies. The comparison of simulators is a complex problem, and typically inaccurate due to the large number of parameters. SimpleScalar provides a toolbox of simulation components as well as several simulators. Each simulator interprets executables compiled by *gcc* version 2.6.3 for a virtual instruction set. SimpleScalar optimizes performance and flexibility by reducing design time and maximizing design exploration. The most complicated and detailed simulator used in the SimpleScalar toolset is *sim-outorder*, which supports out-of-order execution, with a five-stage pipeline. SimpleScalar outputs CPI, IPC and instruction count, along with other processor statistics. Wattch is an architecture simulation tool built for analyzing and optimizing processor power consumption. Brooks *et. al.* quantified power consumption of all major units of the processor and parameterized them. Northern *et. al.* integrated these parameters into the GA, where they were able to perform comparative evaluations between different solutions during the search process [19]. Chapter 5 illustrated the prototype version of the simulation system on two benchmarks. The results illustrate that the system level tool was able to reduce power and increase performance.

GA research has provided a source of techniques that can be used to improve the efficiency of a search and the results obtained. Unfortunately, no guiding metric is available to select techniques for any particular application. Therefore Chapter 5 investigates techniques with the aim of improving GA performance for this application.

In Chapter 6, a technique was presented to exploit the multi-solution nature of the GA search mechanism to provide trade-off information to the system designer. The information, presented in the form of Pareto-charts, illustrates the highest performance solutions across the power range. This enables the designer to select the solution that best meets the implementation on the minimization of a single parameter. In addition, the Pareto-charts are useful in examining the nature of the solution space illustrating the large effect that performance increases can have on power consumption. This tool is the first system-level performance-power optimization tool to present weighted Pareto trade-off information as part of the optimization process, exploiting the inherent characteristics of the GA search technique.

7.4 Conclusions

The main contributions of this work are the following:

1. Methodology for exploring embedded system architectures [Chapter 5]: Issues related to processor architecture and organization, are addressed in an evolutionary approach for programming processors. An application of a simple genetic algorithm in the exploration of a configurable processor is used in the process. With this methodology we are able to optimize embedded processors for practical applications to a much larger extent [19].

2. Multi-objective analysis for configuring embedded system architectures [Chapter 6]: An efficient multi-objective genetic algorithm that generates a set of alternative solutions and indicates the best power/performance tradeoff was developed. These alternative solutions are expressed as non-dominated

points (*i.e.*, a solution is dominant over another only if it has superior performance in all criteria).

3. Sensitivity analysis for embedded system optimization [Chapter 6]: Based on experimentation with practical applications, we have determined a heuristic for training the genetic algorithm to better represent the configuration problem. The genetic algorithm replaces the less sensitive parameter feature with their average [88].

7.5 Future Work

On the technology side, the problems of accurate and fast high-level power estimation have been discussed. Further refinements to the power analysis module, such as the consideration of capacitance characterization for a particular device technology could improve the accuracy of the analysis.

The application of standard genetic operators, during the search process, could result in some invalid designs with incorrect functionality. Rather than implement these corrupt designs, which increases computation time and reduce the efficiency of the search technique, problem-specific architecture techniques should be incorporated into the GA. This would enable the GA to search the design space more effectively.

The current system employs a sensitivity algorithm that reduces the design space by pruning non-significant parameters. This algorithm can

further be improved by analyzing the linkage between parameters and their effect on the evaluation values.

The Pareto-chart uses the data produced during the GA search and optimization process. This could be further refined by incorporating Multi-Objective GA (MOGA) techniques such as niching and fitness sharing [90]. Such techniques can improve the ability of the GA to fully explore the range of available trade-offs.

The GA has been compared with other search techniques that target optimization of combinational problems. There are many other techniques such as Tabu Seatch, ILP, simulated annealing, hill-climbing, *etc.* The development of combined techniques with the GA as the core search algorithm could provide a set of routines for the implementation of alternative search and optimization techniques.

Finally, in terms of evaluation of the tool, the actual fabrication of the devices designed with the methodology developed in this thesis will enable practical examination of the power reductions and the associated implications for power, performance, and functionality.

BIBLIOGRAPHY

[1] Chang, H., Cooke, L., Hunt, M. Martin, G., McNelly, A., Todd, L., "Surviving the SOC Revolution – A Guide Platform-based Design," Kluwer Academic Publishers, Boston, MA, 1999.

[2] Adve, S., Burger, D., Eigenmann, R., Rawsthorne, A., Smith, M., Gebotys, C. Kandemir, M., Lilja, D., Choudhary, A., Fang, J., Yew, P.C., "The Interaction of Architecture and Compilation Technology for High-Performance Processor Design," IEEE Computer Magazine, Vol. 30, No. 12, pp.51-58, Dec. 1997.

[3] Glossner, J., "Trends in Compilable DSP Architecture," Keynote Address, IEEE Int'l. Workshop on Signal Processor Systems (SIPS-2000), Lafayette, LA, October 2000.

[4] Catthoor, F., "Energy-delay Efficient Data Storage and Transfer Architectures: Circuit Technology Versus Design Methodology Solutions," Proc. 1st ACM/IEEE Design and Test in Europe Conf., Paris France, Feb 1998.

[5] Burger, D., Goodman, J., and Kagi, A., "Limited Bandwidth to Affect Processor Design," IEEE Micro, 17(6), November/December 1997.

[6] Faraboshi, P., Desoli, G. Fisher, "The Latest Work in Digital Signal and Multimedia," IEEE Signal Processing Magazine, pp.59-85 March 1998

[7] Igura, H., Narita, S., Naito, Y., Kazamma, Kuroda, I., Motomura, Amashina, M, "An 800 MOPS 110 mW 1.5V Parallel DSP for Mobile Multimedia Processing," Proc. IEEE Int. Solid State Circ. San Francisco, CA, pp.292-293, Feb. 1998.

[8] Kuroda, I., Nishitani, T, "Multimedia Processors", Proceedings of the IEEE, Vol. 86, No. 6, pp. 1203, June, pp.1203-1221, June 1998.

[9] Lee, W., Landman, P., Barton, B.,Abiko, S.T. Takahashi et al., "A 1V DSP for Wireless Communications", Proc. IEEE Int. Solid-State Circuits Conference, San Franscico, CA, pp.92-93, Feb 1997.

[10] Brooks, D., Tiwari, V., Martonosi, M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," Proceedings of the 27th International Symposium on Computer Architecture, June 2000, Vancouver, BC, pp.83-94.

[11] Ellervee, P., Miranda, M., Catthoor, F., Hemani, A., "System-level Data Format Exploration for Dynamically Allocated Data Structures." The 37th Design Automation Conference (DAC'2000), pp.556-559, Los Angeles, CA, June 2000.

[12] Catthoor, F., de Greef, E., Suytack, S., "Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design," Kluwer Academic Publishers, Norwell, MA, 1998.

[13] Texas Instruments TMX320C6201 DSP Data Book, Texas Instruments, Dallas, TX, 1998.

[14] TriMedia TM1000 Data Book, Philips Semiconductors, version 1.0, Sunnyvale, CA, 1997.

[15] Pentium P-III Data Book, Intel Corporation, Santa Clara, California, 1999.

[16] Sudharsanan, S., "MAJC-5200: A High Performance Microprocessor for Multimedia Computing," Lecture Notes in Computer Science (PDIVM/IPDPS 2000), Vol. 1800, pp.161-170, May 2000.

[17] Panda, P., Dutt, N., Nicolau, A., "Memory Issues in Embedded Systems-on-Chip," Kluwer Academic Publishers, Boston, MA, 1999.

[18] Skadron, K., Ahuja, P., Martonosi, M., Clark, D., "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-offs and Simulation Techniques," IEEE Transactions on Computers, Vol. 48, No. 11, pp. 1260-1281, Nov. 1999.

[19] Northern, J., Shanblatt, M., "An Evolutionary Approach to Configuring a Portable Embedded System," IWSOC, Calgary, Alberta, Canada 2003.

[20] Srinivas, D., Sharad, M., " A Survey of Optimization Techniques Targeting Low Power VLSI Circuits," Proceedings of the 32nd ACM/IEEE conference on Design automation conference, San Francisco, CA, pp. 242 – 247, 1995.

[21] Burger, D., Austin, T., "The SimpleScalar Tool Set, Version 2.0," Computer Architecture News, pp. 13-25, June 1997.

[22] Price, C., "MIPS IV Instruction Set, revision 3.1," MIPS Technologies, Inc. Mountain View, CA, January 1995.

[23] Austin, T., Mudge, T., Grunwald, D., "PowerAnalyzer for Pocket Computers," DARPA Research Proceedings, http://www.eecs.umich.edu/~jringenb/power/pdfs/FINAL-PowerAnalyzer-april-meeting.pdf, April 2001.

[24] Bahar, R. I., Albera, G., Manne, S., "Power Performance Tradeoffs Using Various Caching Strategies," Proc. Of Int'l Symposium on Low-Power Electronics and Design, 1998.

[25] Brooks, D., Martonosi, M., "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," Proc. of the 5th Int'l Symposium on High-Performance Computer Architecture, Jan. 1999.

[26] Kamble, M., Ghose, K., "Analytical Energy Dissipation Models for Low Power Caches," Proc Int'l Symposium on Low Power Electronics and Design, pp. 143-148, Monterey, CA, Aug. 1997.

[27] Su, C., Despain, A., "Cache Designs for Energy Efficiency," In Proc. Of the 28th Hawaii Int'l Conference on Systems Science, 1995.

[28] Montanaro, J., "A 160-MHz, 32-b, 0.5W CMOS RISC Microprocessor," Digital Technical Journal, 9(2):49-62, 1996.

[29] Manne, S., Klauser, A., Grunwald, D., "Pipeline Gating: Speculation Control for Energy Reduction," Proc. Of the 25th Int'l. Symposium on Computer Architecture, pp.132-41, June 1998.

[30] Brooks, D., Tiwari, V., Martonosi, M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," Int'l. Symposium Computer Architecture (ISCA), pp. 83-94, Vancouver, Canada, May 2000.

[31] Chen, R., Irwin, M., Bajwa, R., "An Architectural Level Power Estimator," Power-Driven Microarchitecture Workshop at ISCA 25, 1998.

[32] Palacharla, S., Jouppi, N., Smith, J., "Complexity-Effective Superscalar Processors," Univ. of Wisconsin Computer Science Tech. Report 1328, 1997.

[33] Bishop, B., Kelliher, T., Irwin, M., "The Design of a Register Renaming Unit," In Proc. Of Great Lakes Symposium on VLSI, 1999.

[34] Borah, M., Owens, R., Irwin, M., "Transistor Sizing for Low Power CMOS Circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 15(6):665-71, 1996.

[35] Zimmerman, R., Fitchner, W., "Low-Power Logic Styles: CMOS versus Pass-Transistor Logic," IEEE Journal of Solid-State Circuits, 32(7):1079-90, 1997.

[36] Fair, H., Bailey, D., "Clocking Design and Analysis for a 600MHz Alpha Microprocessor," In ISSCC Digest of Technical Papers, pages 398-399, February 1998.

[37] Bowhill, W. J., *et. al.*, "Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," Digital Technical Journal, 7(1):100-118, 1995.

[38] Wilton, S., Jouppi, N., "An Enhanced Access and Cycle Time Model for On-chip Caches," In WRL Research Report 93/5, DEC Western Research Laboratory, 1994.

[39] Roberts, F., "Applied Combinatorics," Prentice Hall, 1984.

[40] Torn, A., "Stochastic Methods for Essentially Unconstrained Multimodal Nonlinear Optimization Problems," Slides for Global Optimization, http://www.abo.fi/~atorn/ProbAlg/Page510.html.

[41] Ye, T., Kalyanaraman, S., "An Adaptive Random Search Algorithm for Optimizing Network Protocol Parameters," Technical Report, Renesselaer Polytechnic Institute Computer Science Department, June 2001.

[42] Optimization Technology Center, Online: http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/discrete/integerprog/section2_1_1.html.

[43] Hwang, E., "FSMD Functional Partitioning for Low Power," Design, Automation and Test in Europe, p.22-28, Munich, Germany, March 1999.

[44] Ishihara, T., Yasuura, H., "Voltage Scheduling Problem for Dynamically Variable-Voltage Processors," Proc. 1998 Int'l Symposium Low Power Electronics and Design, ACM Press, pp. 197-202, New York, 1998.

[45] Kernighan, W., Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Technical Journal, 49, pp. 291-307, 1970.

[46] Fiduccia, C. M., Mattheyses, R. M., "A Linear-Time Heuristics for Improving Network Partitions," Proc. of Design Automation Conference, Las Vegas, NV, pp. 175-181, 1982.

[47] Goldberg, M. K., Burnstein, M., "Heuristic Improvement Technique for Bisection of VLSI Networks," Proc. IEEE Int. Conf. on Computer Design, Port Chester, New York, pp. 122-125, 1983.

[48] Kring, C., Newton, A. R., "A Cell-Replicating Approach to Mincut-Based Circuit Partitioning," Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara, CA, pp. 2-5, 1991.

[49] Hendrickson, B., Leland, R., "A Multi-level Algorithm for Partitioning Graphs," Technical Report SAND93-1301, Sandia National Laboratories, Albuquerque, NM 87185-1110, 1993.

[50] Wei, Y., Cheng, C., "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning," Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara, CA, pp. 298-301, 1989.

[51] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., Teller, E., "Equations of State Calculations by Fast Computing Machines," J. Chem. Phys. 21, pp. 1087- 1092, 1958.

[52] Pincus, M., "A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems," Oper. Res. 18, pp. 1225-1228, 1970.

[53] Kirkpatrick, S., Gerlatt, Jr., C. D., Vecchi, M. P., "Optimization by Simulated Annealing," Science 220, pp. 671-680, 1983.

[54] Sun, W., Sechen, C., "A Loosely Coupled Parallel Algorithm for Standard Cell Placement," Proc. IEEE Int. Conf. on Computer-Aided Design, San Jose, CA, pp. 137-144, 1994.

[55] Sato, S., "Simulated Quenching: A New Placement Method for Module Generation," Proc. IEEE Int. Conf. on Computer-Aided Design, San Jose, CA, pp. 538-541, 1997.

[56] Darwin, C., "On the Origin of Species," Available FTP ftp://sunsite.unc.edu Directory: /pub/docs/books/Gutenberg/etext98/ File: otoos10.txt.

[57] Fei, Y., Jha, N., "Functional Partitioning for Low Power Distributed Systems of Systems-on-a-chip," DARPA Grant 2002.

[58] Martin, R. S., Knight, J. P., "Power-Profiler: Optimizing ASICs Power Consumption at the Behavioral Level," Proc. IEEE Design Automation Conference, pp.42-47, San Francisco, CA, 1995.

[59] Bright, M. S., "Evolutionary Strategies for the High-Level Synthesis of VLSI-Based DSP Systems for Low Power," Ph.D. Dissertation, University of Wales, 1998.

[60] Rechenberg, I., "Evolutionsstrategie-Optimierung technischer Systeme nach Prinzipien der biologischen Evolution," Stuttgart, Frommann-Holzboog, 1973.

[61] Holland, J., "Adaptation in Natural and Artificial Systems," University of Michigan Press, 1975.

[62] Whitley, D., "An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls," Journal of Information and Software Technology, (43), pp. 817-831, 2001.

[63] Goldberg, D. E., "Genetic Algorithms in Search, Optimization and Machine Learning," Reading, MA, Addison-Wesley Publishing Co. Inc., 1989.

[64] V. Schnecke, O. Vornberger, "Genetic Design of VLSI-Layouts," Procs. First IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA 95, Sheffield, U.K., pp. 430-435, Sep 1995.

[65] Cavicchio, D. J., "Adaptive Search Using Simulated Evolution," Ph.D. Thesis, University of Michigan, Ann Arbor, MI, 1970.

[66] Mitchell, M., "An Introduction to Genetic Algorithms," Cambridge, MA, MIT Press, 1996.

[67] Teukolsky, S. A., Vettering, W. T., Press, w. H., Flannery, B. P., (Eds.), "Numerical Recipes in C," 2nd Ed., Cambridge, UK, Cambridge University Press, 1995.

[68] Beasly, D., Bull, D. R., Martin, R. R., "An Overview of Genetic Algorithms: Part 1, Fundamentals," University Computing, 1993, 15(2), pp. 58-69.

[69] De Jong, K. A., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems," Ph.D. Thesis, University of Michigan, Ann Arbor, MI, 1975.

[70] Davis, L., Steenstrup, M., "Genetic Algorithms and Simulated Annealing: An Overview," Genetic Algorithms and Simulated Annealing, Morgan Kaufman, LA, USA, 1987.

[71] Beasley, D., Bull, D. R., Martin, R. R., "An Overview of Genetic Algorithms: Part 2, Research Topics," University Computing, 15(4), pp. 170-181, 1993.

[72] Baker, J. E., "Adaptive Selection Methods for Genetic Algorithms," Proc. 1st Int. Conf. On Genetic Algorithms and Their Applications, PA, USA, pp. 101-111, July 24-26, 1985.

[73] Holland, J. H., "Adaptation in Natural and Artificial Systems," 2nd ed., MIT Press, Cambridge, MA, 1992.

[74] Davis, L., "Adapting Operator Probabilities in Genetic Algorithms," Proc. 3rd Int'l Conference on Genetic Algorithms, CA, June 1989, pp. 61-69.

[75] Goodman, E., "An Introduction to GALOPPS--The 'Genetic Algorithm Optimized for Portability and Parallelism' System," Technical Report GARAGe 96-07-01, Michigan State University, 1996.

[76] Baker, J. E., "Reducing Bias and Inefficiency in Selection Algorithm," Proc. 2nd Int. Conf. On Genetic Algorithms, 1987.

[77] Givargis, T., Vahid, F., "Parameterized System Design," International Workshop on Hardware/Software Codesign (CODES), San Diego, May 2000.

[78] Vahid, F., Givargis, T., "The Case for a Configure-and-Execute Paradigm," International Workshop on Hardware/Software Codesign (CODES), Rome, May 1999.

[79] Arslan, T., Horrocks, D. H., Ozdemir, E., "Structural Cell-based VLSI Circuit Design Using a Genetic Algorithm," Proc. IEEE Int. Symposium on Circuits and Systems, Atlanta, GA, vol. 4, pp. 308-311, 1996.

[80] Arslan, T., Horrocks, D. H., Ozdemir, E., "Structural Synthesis of Cell-based VLSI Circuit Design Using a Multi-Objective Genetic Algorithm," Proc. IEE Electronic Letters, vol. 32, no. 7, pp. 651-652, March 1996.

[81] Gwee, B. H., and Lim, M. H., "Plyominoes Tiling by a Genetic Algorithm," Computational Optimization and Applications 6, Kluwer Academic Publishers, pp.273-291, Massachusetts, 1996.

[82] Ebsen, H., and Kuh, E. S., "Design Space Exploration Using the Genetic Algorithm," Proc. IEEE Int. Symposium on Circuits and Systems, pp. 500-503, Atlanta, GA, May 1996.

[83] Bright, M. S., and Arslan, T., "Transformational-based Synthesis of VLSI Based DSP Systems for Low Power Using a Genetic Algorithm," IEEE Int. Symposium on Circuits and Systems, Monterey, CA, 1998.

[84] Martin, R. S., and Knight, J. P., "Optimizing Power in ASIC Behavioral Synthesis," IEEE Design and Test of Computers, pp. 58-70, 1996.

[85] Gajski, D. D., Ramachandran, L, "Introduction to High-Level Synthesis," IEEE Design and Test of Computers, pp. 45-54, 1994.

[86] Roth, A., online at http://www.cis.upenn.edu/~amir/cis501-01, CIS 501: Introduction to Computer Architecture, University of Pennsylvania, 2001.

[87] Standard Performance Evaluation Corporation, online at http://www.spec.org/.

[88] Northern, J., Shanblatt, M., "A Parameter Sensitivity Approach to Configuring an Embedded System," Poster Presentation at GECCO, Chicago, 2003.

[89] Amdahl, G., "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

[90] Murata, T., Ishibuchi, H., "MOGA: Multi-Objective Genetic Algorithms," Proceedings of 1995 IEEE International Conference on Evolutionary Computation, Perth, Australia, pp.289-294, November, 1995.

[91] Lau, T., Tsang, E., "Applying a Mutation-Based Genetic Algorithm to Processor Configuration Problems," International Conference on Tools with Artificial Intelligence, pp. 17-24, 1996.