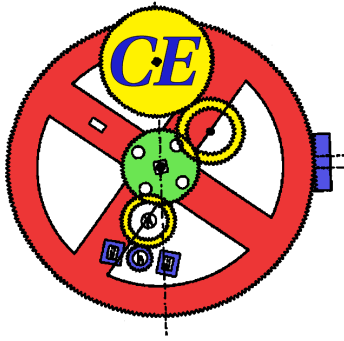


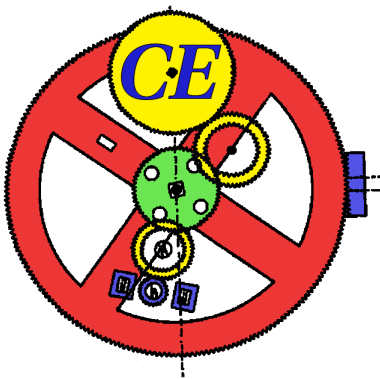


MSc THESIS

Heuristic Search for Defect Tolerant Adaptive Multiprocessor Arrays

Vasileios Vasilikos

Abstract



CE-MS-2011-07

In this paper, new heuristic-search methods and algorithms are presented for enabling highly efficient and adaptive, defect-tolerant multiprocessor arrays. We consider systems where a homogeneous multiprocessor array lies on top of reconfigurable interconnects which allow the pipeline stages of the processors to be connected in all possible configurations. Considering the multiprocessor array partitioned in substitutable units at the granularity of pipeline stages, we employ a variety of heuristic-search methods and algorithms to isolate and replace defective units. The proposed heuristics are designed for off-line execution and aim at minimizing the performance overhead necessarily introduced to the array by the interconnects' latency. An empirical evaluation of the designed algorithms is then carried out, in order to assess the targeted problem and the efficacy of our approach. Our findings indicate this to be a NP-complete computational problem, however, our heuristic-search methods can achieve 100% accuracy in finding the optimal solution among 10^{19} possible candidates within 2.5 seconds. Alternatively, they can provide near-optimal solutions at an accuracy which consistently exceeds 70% (compared to the optimal solution) in only 10^{-4} seconds.

Heuristic Search for Defect Tolerant Adaptive Multiprocessor Arrays

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Vasileios Vasilikos
born in Thessaloniki, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Heuristic Search for Defect Tolerant Adaptive Multiprocessor Arrays

by Vasileios Vasilikos

Abstract

In this paper, new heuristic-search methods and algorithms are presented for enabling highly efficient and adaptive, defect-tolerant multiprocessor arrays. We consider systems where a homogeneous multiprocessor array lies on top of reconfigurable interconnects which allow the pipeline stages of the processors to be connected in all possible configurations. Considering the multiprocessor array partitioned in substitutable units at the granularity of pipeline stages, we employ a variety of heuristic-search methods and algorithms to isolate and replace defective units. The proposed heuristics are designed for off-line execution and aim at minimizing the performance overhead necessarily introduced to the array by the interconnects' latency. An empirical evaluation of the designed algorithms is then carried out, in order to assess the targeted problem and the efficacy of our approach. Our findings indicate this to be a NP-complete computational problem, however, our heuristic-search methods can achieve 100% accuracy in finding the optimal solution among 10^{19} possible candidates within 2.5 seconds. Alternatively, they can provide near-optimal solutions at an accuracy which consistently exceeds 70% (compared to the optimal solution) in only 10^{-4} seconds.

Laboratory : Computer Engineering
Codenummer : CE-MS-2011-07

Committee Members :

Advisor: Ioannis Sourdis, CE, Chalmers University of Technology

Advisor: Christos Strydis, CE, TU Delft

Chairperson: Koen Bertels, CE, TU Delft

Member: Georgi Gaydadjiev, CE, TU Delft

Member: Jan van der Lubbe, ISPL, TU Delft

*This work is dedicated to my parents, who supported me at every
step of the way*

Contents

List of Figures	x
------------------------	----------

Acknowledgements	xi
-------------------------	-----------

1 Introduction	1
1.1 Problem statement	1
1.2 Thesis objectives	3
1.3 Thesis outline	4
2 Background	5
2.1 Computer Architecture	5
2.1.1 Instruction Set Architecture	5
2.1.2 Pipelining	6
2.1.3 Hazards	6
2.1.4 Typical processor pipeline	7
2.2 Fault Tolerance	7
2.3 Combinatorics	9
2.3.1 Permutations	10
2.3.2 Combinations	10
2.4 Heuristic Search	12
2.4.1 Complete state formulation	12
2.4.2 Heuristics	13
2.4.3 Local state space search	13
2.4.3.1 State space & landscape	14
2.4.3.2 Hill Climbing	15
2.4.3.3 Simulated Annealing	16
2.4.4 Genetic algorithms	17
2.5 Conclusions	17
3 Problem Statement	21
3.1 System setup	21
3.1.1 The multiprocessor array	21
3.1.2 Granularity of substitutable units	21
3.2 Problem definition	23
3.2.1 Tolerance & Reconfiguration	23
3.2.2 Interconnect impact	24
3.2.3 Re-definable constraints	24
3.3 System overview	25
3.4 Related Work	26

3.5	Conclusions	27
4	Proposed Approach	29
4.1	Choosing the problem domain	29
4.1.1	Combinatorial analysis	29
4.2	Problem specifics	30
4.2.1	Bottleneck columns	30
4.2.2	Problem size	30
4.2.2.1	External constraint impact on size	31
4.3	Heuristic functions	32
4.3.1	Manhattan distance	33
4.3.2	Throughput loss	35
4.3.3	Bubble stage impact	37
4.3.3.1	Simulating processors in Processor Designer®	38
4.3.3.2	Formal definition of the Bubble Impact heuristic	44
4.4	External constraints	45
4.5	Heuristic search algorithms	46
4.5.1	Greedy search	46
4.5.2	Exhaustive search	47
4.5.2.1	Look-up tables	50
4.5.3	Landscape investigation	52
4.5.4	Hybrid search using Simulated Annealing	53
4.5.5	Partial search	55
4.5.6	Seeded search	55
4.5.7	Genetic search	56
4.5.8	Random search	59
4.6	Conclusions	59
5	Evaluation	61
5.1	Experimental setup	61
5.1.1	Input sizes	61
5.1.2	Benchmark configurations	62
5.1.3	Measured quantities	62
5.2	Evaluation results	64
5.2.1	State space size	64
5.2.2	Memory use	64
5.2.3	Execution time	65
5.2.3.1	Tuning impact	66
5.2.3.2	External constraint impact	67
5.2.3.3	Heuristic function impact	68
5.2.3.4	Algorithm comparison	68
5.2.4	Solution quality & accuracy	71
5.2.4.1	Tuning impact	71
5.2.4.2	External constraint impact	72
5.2.4.3	Heuristic function impact	74

5.2.4.4	Algorithm comparison	75
5.2.5	Simulated multiprocessor array performance	82
5.2.5.1	The “special” case of bursty processing demand	84
5.3	Conclusions	85
6	Conclusions	87
6.1	Conclusions on the evaluation results	87
6.2	Thesis contributions	88
6.3	Future work	89
	Bibliography	93
A	Appendix: Extensive Evaluation	95
A.1	Landscape plots	95
A.2	Execution time plots	97
A.3	Solution quality plots	105
A.4	Solution accuracy plots	113

List of Figures

1.1	Defect isolation & replacement	2
1.2	Two different reconfigurations on the same defect set	3
2.1	Typical processor pipeline (dotted lines indicate forwarding wires)	8
2.2	Transistor consistency trend [4]	8
2.3	Ageing effects trend [4]	9
2.4	Generating all Permutations	11
2.5	Generating all Combinations	12
2.6	General local search algorithm	14
2.7	A hypothetical landscape	15
2.8	Hill Climbing search algorithm	15
2.9	Simulated Annealing search algorithm	17
2.10	Modified Simulated Annealing search algorithm	18
2.11	Genetic search algorithm	18
3.1	A 4x5 multiprocessor array	22
3.2	A 4x5 multiprocessor array with 2 defective blocks	22
3.3	A 4x5 decoupled multiprocessor array	23
3.4	A 4x5 multiprocessor array with 2 defects and 3 working processors	24
3.5	A possible design of the interconnect (dots indicate switching elements)	25
3.6	Overview of the system	26
4.1	A 4x5 defective multiprocessor array, with bottleneck columns marked	31
4.2	An example processor x and it's evaluation $h_{MD}(x)$	34
4.3	An example processor x and it's evaluation $h_{TL}(x)$	36
4.4	Example reconfiguration & respective pipelined interconnect implementation	37
4.5	Instruction mix of all micro-benchmarks	42
4.6	Bubble impact evaluation for all micro-benchmarks	43
4.7	Approximation of cycle impact per bubble at each injection point	44
4.8	An example processor x and it's evaluation $h_{BI}(x)$	45
4.9	Comparative example of two configurations using $C_L(X, A)$ with $L = 1$	46
4.10	Greedy search algorithm	47
4.11	Representation of complete states composed of column states	48
4.12	BruteForce algorithm	49
4.13	Retrieving a column state from the LUT with $k = 2$	50
4.14	FastBruteForce algorithm	51
4.15	Landscape of a 4x5 array with $k_{max} = 2$	52
4.16	Hybrid algorithm	54
4.17	Partial algorithm	56
4.18	Reverse lookup of a column from a configuration X_G with $k = 2$	57
4.19	Seeded algorithm	57

4.20	Crossover and mutation, with $M = 4$	58
4.21	Random algorithm	60
5.1	The 12 different benchmark configurations	62
5.2	The algorithm parameters for each tuning	63
5.3	State space size	65
5.4	Memory consumption	66
5.5	Acc-Off-MD vs Fast-Off-MD (execution time)	67
5.6	Fast-Off-TL vs Fast-On-TL (execution time)	67
5.7	Acc-Off-MD vs Acc-Off-TL vs Acc-Off-BI (execution time)	69
5.8	Execution time comparison, Acc-Off-BI configuration	70
5.9	Acc-Off-MD vs Fast-Off-MD (heuristic value)	71
5.10	Acc-Off-MD vs Fast-Off-MD (accuracy)	72
5.11	Acc-Off-TL vs Acc-On-TL (heuristic value)	73
5.12	Acc-Off-TL vs Acc-On-TL (accuracy)	73
5.13	Fast-Off-MD vs Fast-Off-TL vs Fast-Off-BI (accuracy)	74
5.14	Solution quality (heuristic value) comparison, Fast-Off-MD configuration .	76
5.15	Solution accuracy comparison, Fast-Off-MD configuration	77
5.16	Solution quality (heuristic value) comparison, Fast-Off-TL configuration .	78
5.17	Solution accuracy comparison, Fast-Off-TL configuration	79
5.18	Solution quality (heuristic value) comparison, Fast-Off-BI configuration .	80
5.19	Solution accuracy comparison, Fast-Off-BI configuration	81
5.20	Heuristic accuracy: Bubble Impact IPC vs Simulated IPC	83
5.21	<i>Genetic</i> vs <i>Greedy</i> vs <i>Ideal</i> IPC comparison	84
5.22	<i>Genetic</i> vs <i>Greedy</i> vs <i>Ideal</i> cycles (bursty demand)	86

Acknowledgements

I would like to thank Ioannis Sourdis and Christos Strydis, for their guidance and help during the research and work for this thesis. Many of the ideas in this thesis came after long discussions with them, and the final outcome of this work would not be nearly as interesting if not for their constant flow of ideas and suggestions.

I would also like to thank our secretary, Lidwina, and our sysadmins, Erik & Eef, for their support during the work of this thesis as well as during the study program.

I am very, very grateful to my parents and my sister, that continuously supported me in my studies away from home. Their love and guidance helped me make some of the most important decisions in my life so far and made it possible for me to pursue my desires in my field of study. I love you!

Life in Delft was not just studies. It was, if you can believe it, some fun too! I owe that part to my colleagues here in TU Delft. Nikos, Antonis (both of you!), Giorgos (all of them!), Stavros (levie...), Dimitris (both of them, you see...), many thanks for making my time in Delft fun!

Finally, I would like to thank the person that made the last 7 years of my life the best that they could possibly be: my girlfriend Valia. Thank you for enduring these 3 years that I have been away, for being with me no matter what. I could not have made it without you :-)

Vasileios Vasilikos
Delft, The Netherlands
July 15, 2011

Introduction

As semiconductor technology scales, chips are becoming ever less reliable [4], prominent reasons for this phenomenon being the sheer number of transistors on a given silicon area and their shrinking device features. Future chips are expected to have an increasing number of design, manufacturing and lifetime defects. As a consequence, defect tolerance is becoming an increasingly important issue in Systems-on-Chip (SoCs). If this trend persists, it will be one of the most important aspects of future SoCs and embedded systems in general, especially since embedded systems are nowadays becoming the dominant class of designed systems.

Besides, multiprocessor arrays are becoming more and more popular as the architectural design choice for a system's processing unit; that is due to the difficulty to further scale uniprocessor performance and the increasing number of resources being integrated on a single chip [25]. In response to this, software development is becoming more and more oriented towards multi-threaded applications that can exploit any parallelism in the tasks they perform. This, in turn, means that defects in multiprocessor arrays capable of knocking out entire processors have a direct impact on that system's performance. It is with these observations in mind that the motivation for a defect-tolerant multiprocessor array design becomes clear. A multiprocessor array that can maintain high reliability in the presence of defects can prove to be the major architectural demand for future embedded systems and SoCs.

In the remainder of this chapter we will give a brief problem statement, present the thesis goals and give the outline for the rest of the thesis.

1.1 Problem statement

The problem addressed in this paper is the challenges that emerge from defining a multiprocessor array with defect-tolerance features. We assume arrays of standard general-purpose processors (GPPs) for the evaluation process, but the approach is designed to be parameterizable and modular, in order to be easily employed for other systems as well. We define the granularity of a substitutable unit to be at the level of a pipeline stage, and we assume that the multiprocessor array lies on top of a reconfigurable interconnection network which allows a defective unit to be isolated and replaced by a functioning one. The specific implementation details of the interconnects are not being defined, in the sense that we make no specific assumptions about the implementation of the reconfigurable interconnect.

The goal is to provide defect tolerance by **sparing**, that is, defining pipeline stages of the processors in the array as **substitutable units**, as defined in [16], and shown in Figure 1.1a which are isolated and replaced once they become defective. This isolation-and-replacement ability is provided by the reconfigurable interconnect that we assume

between pipeline stages, and shown in Figure 1.1b

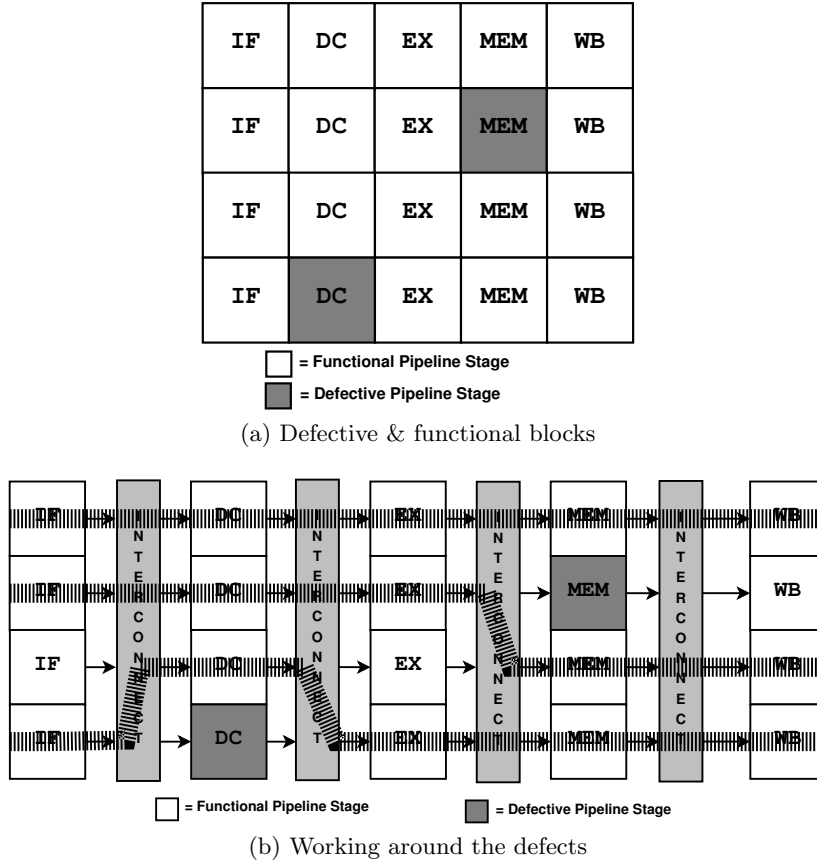


Figure 1.1: Defect isolation & replacement

With this very basic system definition, the problem that arises is that in the presence of defects, there are many reconfiguration options for the reconfigurable interconnect to isolate and replace defective units (as we show in Figure 1.2), and all potentially lead to multiprocessor arrays that have recovered from these defects, albeit with varying degrees of degradation and performance rating. Our goal through this work is to provide efficient exploration mechanisms of the various reconfiguration alternatives so that, depending on the defects and reconfigurable interconnect implementation, the degradation process is as graceful as possible. In other words, given an array of GPPs with reconfigurable interconnects between the pipeline stages, we develop heuristic methods and algorithms to replace defective blocks, maximizing the amount of working processors and keeping the performance penalty to a minimum.

Throughout our work, we consciously make two important assumptions: First, that the reconfiguration process will be taking place **off-line**. This means that instruction execution in the processor array is halted during reconfiguration, and resumes after reconfiguration. The system of course remains online in order to produce a new reconfiguration, but the processors within the array are halted. Second, that after reconfiguration there is **no need for recompilation** of the original source code for the applications

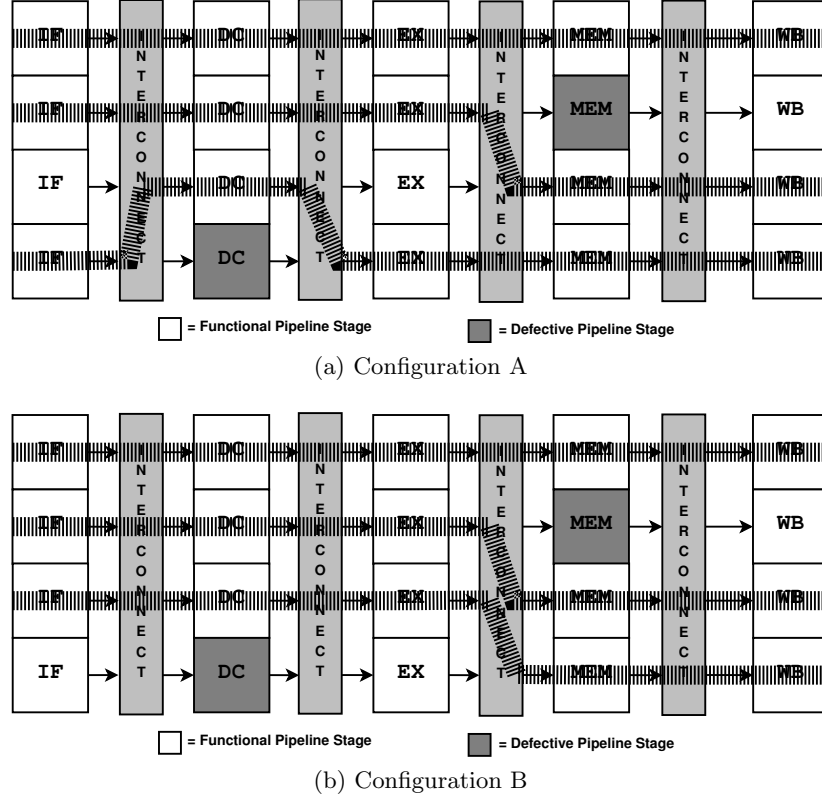


Figure 1.2: Two different reconfigurations on the same defect set

that run on the array. This, in turn, means that the original binaries must be able to be executed on the newly reconfigured array correctly. The system that is thus described presents a wide range of design choices in the fields of processor architecture and reconfigurable interconnects.

1.2 Thesis objectives

With this work we make the following original contributions; we:

- Describe the problem as a *formal computational problem*. This approach allows for a detailed study of the problem's aspects such as its complexity, potential approaches and solution expectations;
- Describe the general concept of “*array performance*” with specific and formal definitions. Through exploration of the architectural issues of a defect-tolerant multi-processor array, define approximations of *performance metrics* (i.e. heuristics), in order to properly evaluate – at the next step – the impact of each reconfiguration choice;
- Propose and implement *search algorithms* that solve the problem by providing the best performance according to the imposed metrics while maintaining correct

functionality of the array architecture. Furthermore, the algorithms have been need to be adjustable for faster execution or higher accuracy; and

- Evaluate our proposed algorithms in terms of the complexity and efficiency of the solutions, in order to provide conclusions suitable for wide adoption in the design of defect-tolerant multiprocessor arrays.

1.3 Thesis outline

The rest of this thesis is organized as follows: In Chapter 2 we provide the necessary background for the concepts of this work and explain briefly why they are needed. In Chapter 3 we present the problem statement at length, where each assumption that we made is clearly defined, and the exact problem is described in a detailed fashion. We also present related research. In Chapter 4 we define our proposed approach in detail and with as formal definitions as possible. In Chapter 5 we perform an empirical evaluation of our approach, present the results of this evaluation and also provide an interpretation thereof. Finally, in Chapter 6 we provide conclusions and discussion about the results and their impact, give potential directions for future work and present the thesis contributions. We also provide in an appendix (Appendix A) the extensive results of our empirical evaluation process.

Background

In this chapter we provide some of the necessary background for this thesis divided into the fields that it relates to. They appear in the remainder of the chapter in the following order: computer architecture in 2.1, in order to define the concepts and quantities that are mentioned when we address the architectural issues discussed in section 1.2. Fault tolerance in section 2.2, where we describe what exactly fault tolerance is and provide the main motivation for fault tolerant system design. Combinatorics in section 2.3 in order to provide formal definitions for some of the concepts used in the proposed approach (in chapter 4) and finally heuristic search in section 2.4 which is the main theoretical domain of our approach.

2.1 Computer Architecture

Computer Architecture is the conceptual design and fundamental operational structure of a computer system [17]. In this section we will define and briefly describe some of the basic concepts of Computer Architecture that are relevant to this thesis, in order to establish reference points and context for later chapters.

2.1.1 Instruction Set Architecture

The term *Instruction Set Architecture* (ISA) refers to the set of instructions that a processor understands. From the ISA one can derive most aspects of a computer's architecture's functionality, as it defines what a processor can and cannot do and how it does it. It is the programmer-visible (at the assembly level) instruction set. The ISA serves as the boundary between software and hardware in a computer system.

Instructions within an ISA can be categorized according to what they perform [17]:

- *Arithmetic/Logical*: Instructions that perform logical or arithmetic operations on integer or logical types.
- *Float*: Instructions that perform operations on floating point types.
- *Data Transfer*: Instructions that perform data transfer operations.
- *Control*: Instructions that perform operations that control the flow of program execution. These can be *conditional*, if they depend on program data, or *unconditional*, if they are executed without any data dependencies.

2.1.2 Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction [17]. In a computer pipeline, each step in the pipeline completes a part of the instruction. Different steps are completing different parts of different instructions concurrently. Each of these steps is called a **pipeline stage**. The stages are connected one to the next and separated by **pipeline registers** to form a pipeline; instructions enter at one end progress through the stages and exit at the other end.

Throughput is the rate at which operations are completed; In computer terms, it is the rate at which instruction execution is completed. Pipelining operations increases throughput.

2.1.3 Hazards

There are situations, called *hazards* [17], that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the program flow.

In order to avoid these hazards a number of techniques have been proposed. Here, three rudimentary techniques will be defined, **stalling**, **forwarding** and **flushing**.

- *Stalling*

The most intuitive way of dealing with the hazards defined above is called *stalling*. This is essentially the freezing of a part of the pipeline until the hazard is cleared; The part that is stalled does not progress through the pipeline stages, whereas the part that is not stalled continues its path down the pipeline, so that the instructions in that part are completed thus clearing the hazard. This technique can be applied to all three classes of hazards.

- *Forwarding*

It should be obvious that stalling the pipeline has an impact on throughput. Ideally, the pipeline should never stall. In order to get closer to this ideal state we can work around some of the hazards, namely Data hazards, by means of *forwarding*. Forwarding is the passing of data from one stage to another when there is a data dependency between instructions in these stages. By adding wires in the pipeline to facilitate this, stalls that are induced by data dependencies are avoided.

- *Flushing*

We can also address some of the stalls that arise from Control hazards. Control hazards that are left otherwise unchecked will result in instructions residing in the pipeline that should not be there. A detection mechanism can be implemented for such instructions that will *flush* them from the pipeline. This technique has a similar impact with stalling in terms of throughput, but has to be applied less often than stalling, giving a better expected throughput than a pipeline that always stalls when detecting Control hazards.

2.1.4 Typical processor pipeline

Here we show the pipeline structure of a typical RISC processor [17]. It is very similar to the classic MIPS R2000 pipeline [27, 17]. It consists of five pipeline stages:

1. *Instruction fetch* (IF)

In this stage the instruction memory is accessed at the location pointed to by the program counter, and the proper instruction is fetched from that memory. The program counter is then updated.

2. *Instruction decode* (DC)

In this stage the instruction from the previous stage is decoded and the operands are read from the register file, which resides in this stage. Unconditional control flow changes are detected here, and the program counter is appropriately updated.

3. *Execution* (EX)

In this stage the appropriate arithmetic or logical operation is performed based on the decoded operation code and the operands retrieved from the previous stage. Conditional branches are calculated here, and the program counter is updated accordingly.

4. *Memory access* (MEM)

In this stage, instructions that need to access the memory perform their memory operation (reading or writing to the memory).

5. *Write back* (WB)

In this stage the resulting data from the instruction, if there are any, are written back in the register file. After this stage the instruction is cleared from the pipeline.

Figure 2.1 shows this pipeline along with the connections needed for forwarding. This pipeline will be used as the reference pipeline for the rest of this thesis.

2.2 Fault Tolerance

Fault tolerant design enables a system to continue operation even after a certain level of internal failure. This usually comes at the cost of extra hardware and/or software that enables the system to retain some or all of its operational capability after detecting failure in part(s) of the system.

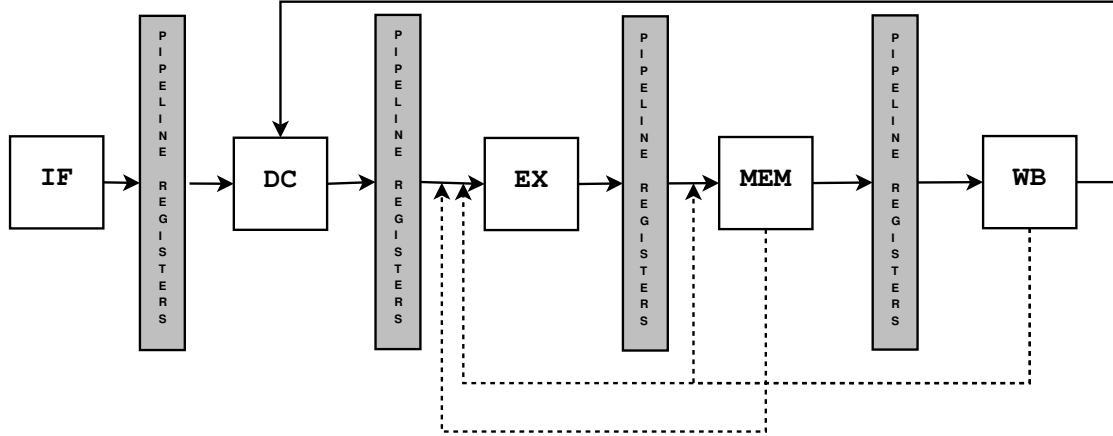


Figure 2.1: Typical processor pipeline (dotted lines indicate forwarding wires)

The term *fault* refers to any and all kinds of malfunctions in a system. We are interested only in those faults that are permanent, called **defects**. Temporary and intermittent faults are not addressed in this thesis. Defects can either be introduced during design, manufacture, or appear later due to ageing. Tolerance refers to the ability to function properly, either by ignoring these defects, working around them, or repairing them.

There has been recent research in this field [4] that investigates the design of reliable systems from unreliable components in the context of the challenges presented by the trends in transistor variability and degradation. In this paper, it is mentioned that in the following years it will become increasingly difficult to guarantee defect-free chips, because the fabrication procedures cannot decrease the defect density to keep up with the increase of the number of transistors in a given area, as seen in Figure 2.2, where it is shown that it is harder to maintain the majority of the transistors operating at a given voltage V_t , and also that systems tend to show a degradation in performance from ageing defects, as shown in Figure 2.3, where the saturation current degradation trend over the years is shown.

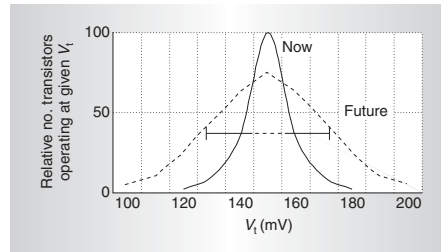


Figure 2.2: Transistor consistency trend [4]

One important concept is that of *regularity* [16]. A system exhibits regularity when it is characterized by repeatable patterns of blocks. This allows any method that applies to one basic block or pattern of blocks to be readily applicable to all other blocks or

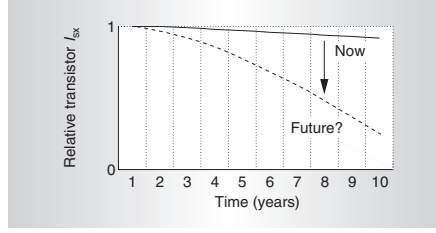


Figure 2.3: Ageing effects trend [4]

patterns. A typical example of such a system is an FPGA. Since FPGAs are based on repeatable Logic Cells, Slices, and Configurable Logic Blocks (CLBs), they present great regularity. When discussing such blocks in the context of redundancy and fault tolerance, we generally refer to these resources as *substitutable blocks* [16]. Since FPGAs show such high regularity, they can be ideal platforms for implementing fault tolerant systems.

As we briefly described in section 1.1, our target system includes a multiprocessor array. Using the above definitions, we will show in chapter 3 how exactly we can model the individual pipeline stages within this array as substitutable resources, in order to facilitate defect tolerance.

We also mentioned in section 1.1 that we study defects at the granularity of pipeline stages. We define here the concept of *sparing* from the field of defect detection, and explain how it apply to this granularity level (of pipeline stages).

Sparing [16] of resources means that we simply mark them as either fully functional or not. We do not make any further distinctions. On the other hand, *matching* (characterizing) resources means further investigation in order to determine exactly which functions are properly performed and which are faulty within the same substitutable unit. Having defined these concepts, we will explain in detail in chapter 3 how we apply **sparing** on substitutable resources at the defined granularity, i.e. that we consider entire pipeline stages as either defective or fully functional.

2.3 Combinatorics

The problem of re-arranging a homogeneous multiprocessor array based on pipeline stages as substitutable blocks means *selecting* the blocks and then *ordering* them. More formally, we need to create a **combination** and a **permutation** of the available substitutable blocks. For this reason, some concepts from the field of combinatorics are needed, and in this section we will provide some background for these concepts. We will show their exact application in detail when we discuss the proposed approach in chapter 4.

Combinatorics in its most general sense is the study of finite or countable discrete structures. For this work, the relevant part of this field is the study of permutations and combinations. Permutations and combinations are part of a group called *arrangements*. This group also includes tuples, partitions, etc. When dealing with arrangements, typically the problems encountered are of the following five categories [23]:

- **Existence:** Are there any arrangements X that conform to the pattern (combination, permutation, etc)?
- **Construction:** If so, can such an X be found quickly?
- **Enumeration:** How many different arrangements X exist?
- **Generation:** Can all arrangements $X_1, X_2, \dots$ be visited systematically?
- **Optimization:** What arrangements maximize or minimize $f(X)$, given an objective function f ?

All of the above are important questions for this thesis, however the latter three (Enumeration, Generation & Optimization) are the most important. Since the arrangements that this thesis covers are permutations and combinations, we will address these questions in the context of these arrangements only. In this section, the issues of **Enumeration** and **Generation** will be covered. The subject of **Optimization** will be addressed in Chapter 4.

2.3.1 Permutations

Definition: Let n be the size of a set S . A **permutation** of a set of elements S is a *unique* ordering of **all** the elements $a_1, a_2, a_3, \dots$ of that set. A k -permutation of a set of elements S is defined as a unique ordering of k elements from the set S , with $k \leq n$.

Enumeration: The amount of total **complete** permutations that are possible on set S is $n!$ [23, 29]. Assuming a subset of S with k elements has been already chosen, the possible k -permutations for this subset are $k!$.

Generation: When dealing with search problems in combinatorics, a systematic way of visiting arrangements is needed. For permutations, there is an algorithm that produces all possible permutations of a set S in a consistent order. The term used for the sorted sequence in which arrangements can appear is called **lexicographical order**. The algorithm that produces permutations in lexicographical order dates back to Narayana Pandita in 14th century India, and has been frequently rediscovered ever since [23, 29]. Figure 2.4 shows a pseudo-code description of this algorithm. This algorithm assumes the set S to be the set of natural numbers $1, 2, 3, \dots, n$, but any set can be mapped on the set of natural numbers, in order to apply the algorithm to it.

2.3.2 Combinations

Definition: A **combination** of k elements from a set S of size n is a subset of S that contains *exactly* k elements, with $k \leq n$. Essentially, a combination is just a subset. Unlike permutations, there is no ordering of the k elements defined for a combination. The term k -*combination* is sometimes used to denote the amount of elements that form a combination.

```

GENERATEPERMUTATIONS.
Input: P (permutation of S), n.
Returns: Next permutation, in-place
  {Let  $P = (a_1, a_2, \dots, a_n)$ }
   $j \leftarrow n - 1$ 
  while  $a_j > a_{j+1}$  do
     $j \leftarrow j - 1$ 
  end while
  { $j$  is the largest subscript with  $a_j < a_{j+1}$ }
   $k \leftarrow n$ 
  while  $a_j > a_k$  do
     $k \leftarrow k - 1$ 
  end while
  { $a_k$  is the smallest integer greter than  $a_j$  to the right of  $a_j$ }
   $swap(a_j, a_k)$ 
   $r \leftarrow n$ 
   $s \leftarrow j + 1$ 
  while  $r > s$  do
     $swap(a_r, a_s)$ 
     $r \leftarrow r - 1$ 
     $s \leftarrow s + 1$ 
  end while
return

```

Figure 2.4: Generating all Permutations

Enumeration: The amount of total k -combinations from S is equal to the binomial coefficient

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}$$

which can be written as factorials $\frac{n!}{k!(n-k)!}$ where $k \leq n$ and which is zero when $k > n$ [23, 29].

Generation: As with permutations, we need to generate combinations in lexicographic order. There is an algorithm that has been described by D. Knuth [23] that generates all k -combinations of a set S with size n in lexicographical order. This algorithm is shown in pseudo-code format in Figure 2.5. This algorithm assumes the set S to be the set of natural numbers $1, 2, 3, \dots, n$, but any set can be mapped on the set of natural numbers, in order to apply the algorithm to it.

After having defined k -combinations, it should be obvious that a k -permutation as defined above is simply a permutation of a k -combination. The **enumeration** of k -permutations would be calculated by observing that there are $k!$ possible permutations

```

GENERATECOMBINATIONS.
Input: C (k-combination of S, sorted ascendingly), k, n.
Returns: Next combination, in-place
  {Let  $C = (a_1, a_2, \dots, a_n)$ }
   $i \leftarrow k$ 
  while  $a_i = n - k + i$  do
     $i \leftarrow i - 1$ 
  end while
   $a_i \leftarrow a_i + 1$ 
  for  $j \leftarrow i + 1$  to  $k$  do
     $a_j \leftarrow a_i + j - i$ 
  end for
  return

```

Figure 2.5: Generating all Combinations

for each k -combination. Hence, from the formulas above, we see that the amount of k -permutations of a set with n elements is equal to

$$\frac{n!}{(n-k)!} = n(n-1)(n-2)\dots(n-k+1)$$

Furthermore, from these definitions we can also deduce that the **generation** of **all** k -permutations of a set S for a given k involves simply applying algorithm in Figure 2.4 to every combination produced by algorithm in Figure 2.5.

2.4 Heuristic Search

Computational problems can be approached by direct algorithms (such as the Dijkstra algorithm [8] for shortest path problems) designed for that specific problem domain. There can be problems however for which a direct algorithm does not exist; in those cases the available choices are either to design a new algorithm for that specific problem from scratch, or to apply some sort of search technique that will approach the solution by abstracting the specifics of the problem definition and will instead treat it as a general computational problem. Naturally there are trade-offs here, and they come usually in the form of either computational complexity, completeness or optimality.

2.4.1 Complete state formulation

A **complete state**, in the context of a computational problem, is a complete description of the environment that the problem resides in [34]. In other words, a complete state is a state in which all possible variables that are defined by the problem have a value assigned to them. For instance, in the problem of finding the sequence of actions that will win a chess game, a complete state would be a representation of the entire chessboard at a particular point in time. As such, we can define the concept of a **goal state**, simply as

a complete state that satisfies the set of restrictions given by the problem that define a solution. We can also define the concept of **state space**, as the set of all possible complete states that can be defined by the problem. The goal state set, by definition, is a subset of the state space.

Using the above definitions, we can formulate almost all computational problems as finding a goal state or the entire goal state set within the state space. We can also define optimization problems as finding a state within the state space that minimizes or maximizes a set of criteria given by the problem definition.

2.4.2 Heuristics

When talking about comparing complete states or testing them for goal state satisfaction, a certain level of knowledge is implied; namely that the information on how to evaluate a complete state based on the real criteria that the problem represents is available. For example, when searching for a shortest path, this information is the actual distance between nodes in that path. This information is not always available and, in some cases, it cannot be properly defined to begin with. For instance, there is no quantitative way of describing exactly how good or bad a chessboard configuration is for a given player in the state space of a chess game.

We need such quantitative values so that we can properly compare complete states when searching for extrema in an optimization problem with search techniques. When the evaluation of a complete state with the actual problem criteria is not possible, either because that information is unobtainable from the problem definition or it is known that such information cannot be defined, or due to problem complexity, we turn to heuristics.

A **heuristic function** is a function that operates on a complete state and returns some value that represents that state's evaluation. Since we do not know if this is close to the actual evaluation value of that state, or if an actual evaluation value can even be defined, the heuristic evaluation of a complete state is used as an indicator, an informal way to judge whether this state is likely to lead the search towards the desired direction or not. In other words, a heuristic function can be described as an "educated guess" of the evaluation of a particular complete state. In the example of the shortest path problem, a heuristic can be the straight line distance between two nodes. In the example of the chessboard, a heuristic could be the amount of own pieces left on the board minus the amount of enemy pieces left on the board.

It should be obvious that the accuracy of this educated guess, i.e. the accuracy of the heuristic function, has great impact on the performance of any heuristic search algorithm. In the case that a heuristic is the only way to evaluate a given state, the quality of the solution will depend mostly on the accuracy of the heuristic.

2.4.3 Local state space search

Once a problem has been formulated using the complete state formulation, the specifics of its definition do not matter. An algorithm can simply search through states in the state space using a specific pattern or strategy, looking for any or all goal states, or global extrema in the case of optimization problems. Usually complete state formulations of problems result in fairly large state spaces, which makes an exhaustive search prohibitive

in terms of time and space (memory) complexity. **Local search** implies that the search is limited within only a subset of the original state space. The search is terminated either when a solution is found or when a pre-set time limit has been exceeded. The pattern or strategy that is used during this search is what defines a specific algorithm, however we can still define a general local search algorithm that describes the general manner in which local search works. Figure 2.6 shows the pseudo-code description of a general local search algorithm for optimization problems.

```

GENERALLOCALSEARCH.
Input: state_space, limit. Returns: Extremum State
{Initialize time and current best state}
time ← 0
extremum ← Strategy(state_space)
while time ≤ limit do
    {Search the state space using strategy()}
    current ← Strategy(state_space)
    if best(current, extremum) = current then
        extremum ← current
    end if
    time ← time + 1
end while
return extremum

```

Figure 2.6: General local search algorithm

2.4.3.1 State space & landscape

As defined above, the state space is a set of complete states. That is, there is no ordering implied by the definition. However, most of the time there is some sort of relationship between the various states of a state space implied by the problem. In its most simple form this is some sort of ordering of the states within a state space. In other words, any given state within a hypothetical ordered state space has a “previous” and a “next” state (except of course for the first and last states in this ordering, in the case of a finite state space).

When the previous and next states of all states are exactly one, we can conveniently place all states along the axis of natural numbers. In the case of optimization problems, we can then **evaluate** each state to find its associated cost (or profit). This would result in a curve that describes what is informally called the **landscape** of the state space. Figure 2.7 shows a hypothetical landscape.

We can see from Figure 2.7 that a landscape can have one or more local extrema, one global minimum and one global maximum, and it can also have regions where the value remains constant, a “plateau”. These features of a landscape greatly affect the performance of local search algorithms, and they are a determining factor in the selection of the actual search algorithm that will be used for a particular problem.

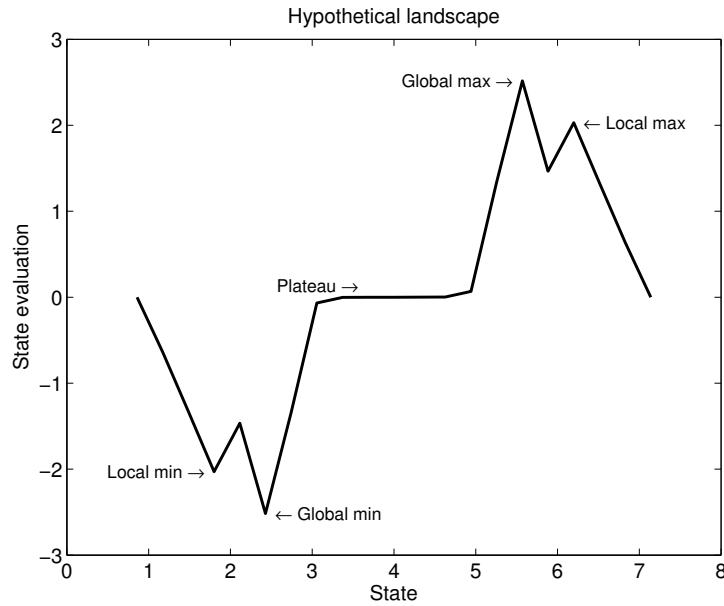


Figure 2.7: A hypothetical landscape

2.4.3.2 Hill Climbing

The simplest method of traversing through the state space is to examine all immediate neighbouring states from the current state, evaluate them and move immediately to the best one. The search is terminated when no better states can be found amongst the immediate neighbours of the current state [34]. Figure 2.8 shows the pseudo-code description of a hill climbing search on a 2-dimensional landscape. Depending on the landscape defined by the state space, this method can have varied performance.

```

HILLCLIMBINGSEARCH.
Input: state_space, limit. Returns: Extremum State
{Initialize time and current best state}
time ← 0
current ← random(state_space)
while time ≤ limit do
    {Search the state space using Hill Climbing}
    left ← getLeftNeighbor(current, state_space)
    right ← getRightNeighbor(current, state_space)
    current ← best(left, current, right)
    time ← time + 1
end while
return current

```

Figure 2.8: Hill Climbing search algorithm

In the case of a smooth, almost monotonic landscape with very few plateaus and very few local extrema, a hill climbing technique is very likely to produce a result which is a global extremum. However, very few problems produce such landscapes. More often than not landscapes tend to have many local extrema and plateaus which can trap a naive hill climbing search very quickly, preventing it from progressing very far into the state space, and in effect limit the search within a very narrow window of states.

2.4.3.3 Simulated Annealing

Simulated Annealing is a local search algorithm first proposed in 1983 [22] and 1985 [36] independently. It is a non-deterministic [18] algorithm that searches through the state space in a non-sequential manner. The algorithm attempts to emulate a technique in metallurgy involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

In optimization problems, heat can be loosely translated to entropy, and the cooling procedure is the rate at which entropy impact is reduced. The Simulated Annealing algorithm, at high “temperatures”, will behave more randomly, moving through states against the direction that the heuristic is pointing at. As temperature drops, the direction of the search becomes more and more compliant with the heuristic. The (usually decreasing) function that determines the temperature as a function of time is called the **schedule function** [34]. Figure 2.9 shows the pseudo-code description of a basic Simulated Annealing search algorithm.

The reason behind such a technique is to avoid the drawbacks of hill climbing when searching in state spaces that have lots of local extrema and plateaus. By allowing the search to move towards seemingly worse states than the current one, it can overcome local extrema better and have a higher chance of finding a global extremum and avoid plateaus (in landscapes that have lots of local extrema and plateaus).

The original definition of the Simulated Annealing algorithm implies a drawback: It is possible that it finds a global extremum but due to high entropy it will be discarded for a worse state. Or, more generally, it will return the state at which the search is located exactly when the algorithm reaches its time limit. If a better state was inspected earlier on but discarded due to high entropy, it will not be “remembered” at the end. This drawback can be overcome by slightly modifying the algorithm to have a “memory” of one state. That is, at any given point in time during its execution, the best state that was encountered is always stored. In the end, it will return that state instead of the current one. Figure 2.10 shows this modified version of Simulated Annealing search.

In this modified version of Simulated Annealing, even if a very good state is encountered but discarded, it will be stored as the best state so far in the added “memory slot”. This memory slot is the final output of the algorithm.

It can be shown [12] that for any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended. This theoretical result, however, is not particularly

```

SIMULATEDANNEALINGSEARCH.
Input: state_space, limit. Returns: Extremum State
{Initialize time and current best state}
time ← 0
current ← random(state_space)
while time ≤ limit do
    {Search the state space using Simulated Annealing}
    left ← getLeftNeighbor(current, state_space)
    right ← getRightNeighbor(current, state_space)
    if Schedule(time) = False then
        current ← best(current, left, right)
    else
        current ← worst(current, left, right)
    end if
    time ← time + 1
end while
return current

```

Figure 2.9: Simulated Annealing search algorithm

helpful, since the time required to ensure a significant probability of success will usually exceed the time required for a complete search of the solution space.

2.4.4 Genetic algorithms

In order to bypass the problem of performance degradation of local search techniques in difficult landscapes, a different approach must be considered altogether. This is the reason why techniques such as Genetic Algorithms were developed. They try to mimic the behaviour of natural selection to gradually converge to the optimal solution of an optimization problem.

A Genetic algorithm [11] ignores the landscape of a problem’s state space. It does not traverse it in any manner; instead, it maintains a population of states that are cross-bred to form new states, and there is always the small chance of random mutation to inject new “genes” into the selection pool. The population is inspected after each “generation” or after a certain amount of generations for the best state within the current population.

Selection policy, cross-breeding method, population size, amount of generations and mutation probability are all design choices for a Genetic algorithm. They usually depend on the specifics of a problem, and their impact has been a subject of extended research [11]. Figure 2.11 shows a general genetic algorithm in pseudo-code.

2.5 Conclusions

In this chapter we presented the important and relative background for the concepts of this work. It included background from the fields of computer architecture, fault

```

SIMULATEDANNEALINGMODIFIED.
Input: state_space, limit. Returns: Extremum State
{Initialize time and current best state}
time  $\leftarrow$  0
current  $\leftarrow$  random(state_space)
optimal  $\leftarrow$  current
while time  $\leq$  limit do
    {Search the state space using Simulated Annealing}
    left  $\leftarrow$  getLeftNeighbor(current, state_space)
    right  $\leftarrow$  getRightNeighbor(current, state_space)
    if Schedule(time) = False then
        current  $\leftarrow$  best(current, left, right)
    else
        current  $\leftarrow$  worst(current, left, right)
    end if
    if best(current, optimal) = current then
        optimal  $\leftarrow$  current
    end if
    time  $\leftarrow$  time + 1
end while
return optimal

```

Figure 2.10: Modified Simulated Annealing search algorithm

```

GENETICALGORITHMSEARCH.
Input: state_space, limit, size. Returns: Extremum State
{Initialize time and current best state}
generations  $\leftarrow$  0
population  $\leftarrow$  generateRandomStates(size) {Generates size random states}
while generations  $\leq$  limit do
    evaluate(population)
    breeders  $\leftarrow$  selection(population)
    offspring  $\leftarrow$  crossover(breeders)
    offspring  $\leftarrow$  mutate(offspring)
    population  $\leftarrow$  population + offspring
    generations  $\leftarrow$  generations + 1
end while
return best(population)

```

Figure 2.11: Genetic search algorithm

tolerance, combinatorics and heuristic search, explaining how they apply to the problem that this thesis addresses. These concepts and their application are important for both

the problem definition given in the next chapter, and also for the description of the proposed approach given in [chapter 4](#).

Problem Statement

In this chapter the definition of the problem that this thesis addresses is given by, first, defining the system that the work is carried out on in section 3.1, where we show the environment that we assume our problem to reside in, and define most of the assumptions that we make. A detailed problem description is given in section 3.2, where we explain in detail what issues we address within the defined system. We conclude the chapter by presenting related research in similar problems in section 3.4.

3.1 System setup

In order for the specifics of the problem definition in the following section to make sense, we need to give the proper context by properly defining the system.

3.1.1 The multiprocessor array

The main component of the system is a homogeneous *multiprocessor array*. This array consists of N processors, each of them having M pipeline stages. We denote this as a $N \times M$ array. The processors are independent of each other, meaning that there is no sharing of memory other than perhaps the main off-chip memory, which is in any case not relevant for this problem. Each processor within this array is assumed to have its own internal memory.

It may be typical for such arrays to have $N = 2^k$ amount of processors, but in this case such an assumption does not need to be made; N can have any positive value. However, to keep the results of this work as relevant as possible to industry standards, the amount of processors studied in the evaluation part of this thesis (in Chapter 5) will be $N = 2^k$. Assuming the typical processor described in Figure 2.1 with $M = 5$, a typical multiprocessor array with $N = 4$ (4x5 array) can be seen in Figure 3.1.

3.1.2 Granularity of substitutable units

In a multiprocessor array, there is, in a way, a **latent redundancy** in the system: A part is available in more than one identical copies, namely N copies of each pipeline stage. This fact in itself does not constitute system-level redundancy since each copy of a pipeline stage is essential to the processor that uses it. This hides the redundancy that would otherwise be obvious, for example in the presence of defects at the granularity level of pipeline stages. In other words, since each processor needs all of its pipeline stages in order to function properly, a defect in any of that processor's pipeline stages would render that processor useless. This, in turn, would make the remaining pipeline stages functional but inaccessible copies, thus redundant.

IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB

Figure 3.1: A 4x5 multiprocessor array

Similarly to [15] and [28], we have defined as substitutable unit of our array each pipeline stage of a processor; that is, the unit which can be isolated, when defective, and replaced by another functioning one. The defect-tolerance feature stems from exploiting this redundancy at the pipeline stages whenever a defect arises in a pipeline stage. In Figure 3.2 an example defective array is shown. Despite the fact that there are 2 defects in 2 different processors, rendering both in-operational, it is obvious that there are enough spare parts to create 3 functional processors, due to this latent redundancy. This is achieved by reconfiguring the interconnects between the substitutable units in order to form 3 processors out of the available functioning stages.

IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB
IF	DC	EX	MEM	WB

= Functional Pipeline Stage
 = Defective Pipeline Stage

Figure 3.2: A 4x5 multiprocessor array with 2 defective blocks

The defect granularity that is going to be addressed is that of pipeline stages. Defects can of course occur at a much finer granularity, even at the individual transistor level, but for this thesis the granularity is defined at the pipeline stage level.

3.2 Problem definition

3.2.1 Tolerance & Reconfiguration

As mentioned above, tolerance to manufacturing or ageing defects comes from exploiting the redundancy generated as soon as these defects appear in a multiprocessor array. It is important to define, however, exactly how this can be achieved, even if only at the theoretical level.

A crucial step in the process of making a multiprocessor array defect-tolerant is the **decoupling** of processors and their respective pipeline stages, as shown in Figure 3.3. This, in effect, means that a pipeline stage is just an element of the system that may or may not belong to the processor that it was manufactured for. In other words, a processor may be constructed from any pipeline stage that is available in the array, that is, not only from the ones that were originally designed for that processor.

Conceptually, this is achieved by looking at the multiprocessor array as a set of independent columns, each containing a number of copies of a pipeline stage, and all of them interconnected through some sort of reconfigurable interconnection network. From this point of view, a processor no longer needs to be defined as a hard-wired set of pipeline stages; instead, it can be defined as a flexible path from the first pipeline stage to the last. In the presence of defects, the interconnection network can be **reconfigured** to redefine all processors within the array. This remedies the situation described before, where a defective pipeline stage would result in a completely defective processor. With a reconfigurable interconnect, a defect can simply be ignored and the remaining elements in the multiprocessor array can be fully utilized, instead of remaining unused. It should be noted at this point that we consider each pipeline stage to be fully assigned to **exactly one** processor; i.e. there is *no stage sharing* between processors.

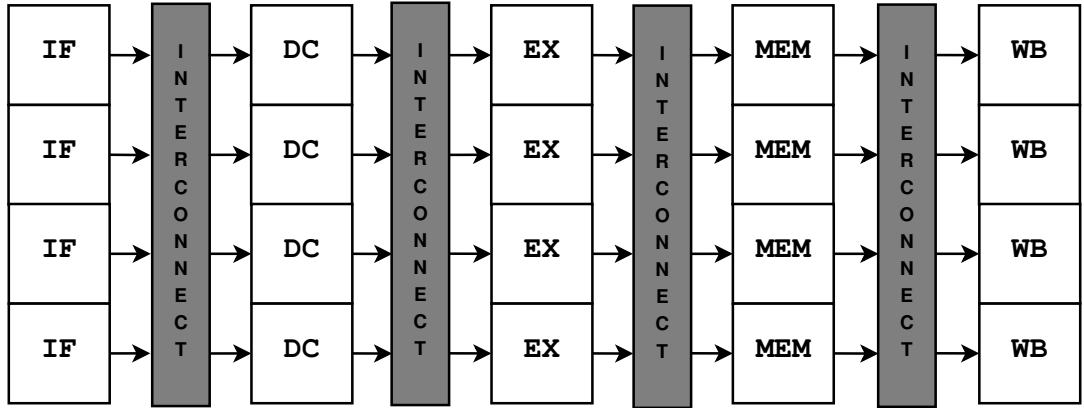


Figure 3.3: A 4x5 **decoupled** multiprocessor array

From this point of view, a processor no longer needs to be defined as a hard-wired set of pipeline stages; instead, it can be defined as a flexible path from the first pipeline stage to the last. In the presence of defects, the interconnection network can be **reconfigured** to redefine all processors within the array. This remedies the situation described before, where a defective pipeline stage would result in a completely defective processor. With a

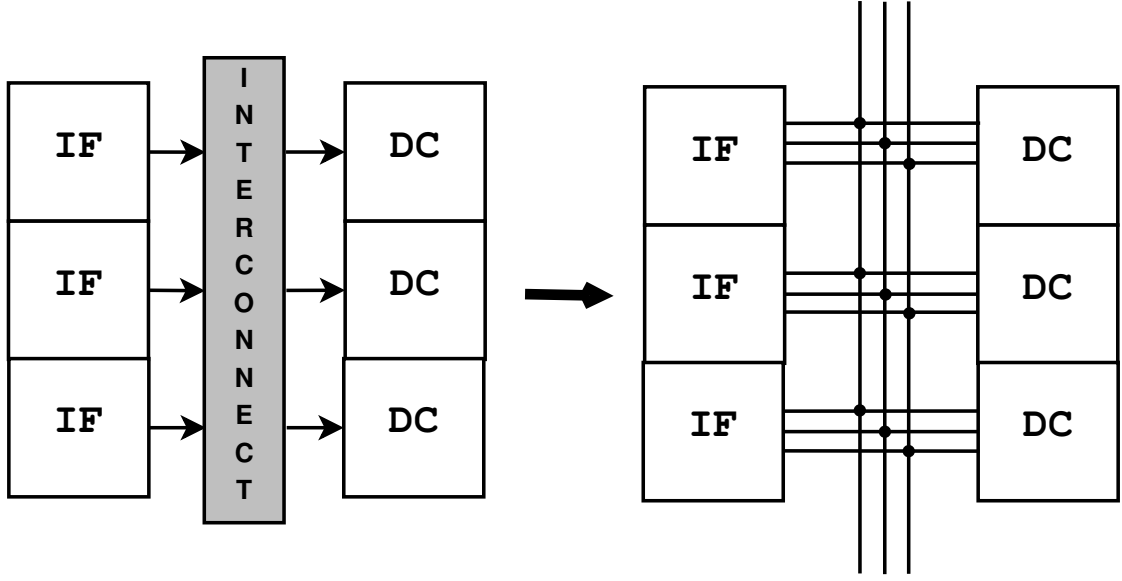


Figure 3.5: A possible design of the interconnect (dots indicate switching elements)

assumptions regarding either the type of processor or the design of the interconnect in this system, so restrictions cannot be hard-coded.

This leads us to a more generic design, where restrictions are viewed as an externally provided set of formal rules that apply to the particular system in question. These restrictions are checked against a candidate configuration. If a particular configuration violates any restriction within the provided set, it cannot be considered a candidate solution. Examples of possible restrictions are:

- Wires within the interconnect are one-direction only
- Stage X has to reside at the same horizontal level as stage Y
- There are at most C wires available at each switch point in the interconnect
- A specific pipeline stage cannot be in a different row than another pipeline stage
- etc

These are just examples to better explain the nature of possible constraints; it is also possible that there are no constraints at all, depending on the system.

3.3 System overview

For this work, we assume that the information on defects within the array is readily available. This information is then passed on to a **reconfiguration manager**, assigned the task of returning a new, valid configuration for the multiprocessor array that will maximize its performance in the presence of these defects. This reconfiguration is defined

as an **off-line** process, meaning that the array is not executing instructions while being re-configured. Execution resumes after the reconfiguration process has been completed. The system of course remains online in order to produce a new reconfiguration, but the processors within the array are halted. It also does not include any sort of recompilation of source code that runs on the array or any change to the original application binaries whatsoever.

The amount of processors N and the number of pipeline stages M are input parameters to the reconfiguration manager as well as the set of constraints. Figure 3.6 shows an overview of the system where these parameters have not been set.

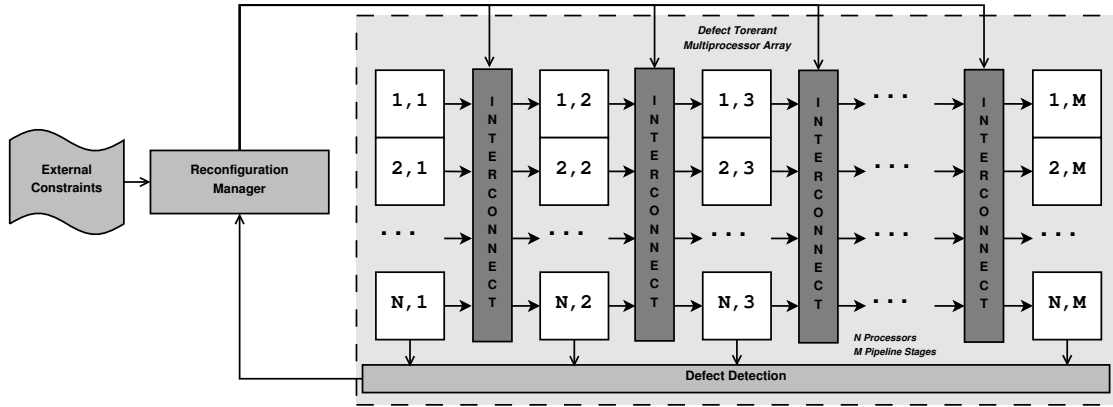


Figure 3.6: Overview of the system

3.4 Related Work

The common way to cope with defects (permanent faults) is to either provide redundant identical components or entire cores (coarse-grain reconfiguration) or fine-grain reconfigurable hardware (i.e. Field-Programmable Gate Arrays – FPGAs). In general, coarser granularity delivers faster circuits and, therefore, better performance while finer approaches are more defect-tolerant.

Some works provide defect tolerance by redundancy at the processor level (redundant cores) [1, 10, 33, 35, 24]. These approaches utilize the redundancy of multi-core systems to provide fault tolerance. The issues addressed involve load re-balancing, fault detection and isolation, memory-access scheduling, execution-thread rescheduling, and performance-degradation evaluations in the presence of faults.

Other works have chosen a granularity of pipeline stages in a multiprocessor array, like in our approach. Such a solution is “StageNet” [15, 13, 14]. As opposed to our approach, though, StageNet uses a crossbar switch to interconnect stages. This effectively means that all different reconfiguration alternatives impose the same delay in the critical path, making the choice of a reconfiguration option redundant in terms of variation in processor performance. Another major difference with the present work is their processor-specific pipeline, in contrast to the generic approach that we work with.

Another approach with a pipeline-stage granularity is the “Core Cannibalization Architecture (CCA)” proposed by Romanescu and Sorin [28]. In their approach they extended the problem to address some of the performance overhead caused by the interconnect. They proposed both a clock-frequency decrease approach by lengthening interconnect wires to get spare parts, but also a pipelined-interconnect approach in order to maintain clock frequency. However, Romanescu and Sorin do not propose a generic or systematic approach for finding an efficient configuration to maximize the functionality and performance of the multiprocessor array.

There has also been research aimed at exploiting the inherent redundancy of super-scalar cores [31, 5, 6]. Techniques used include investigating fault tolerance by exploiting the redundancy in microprocessor array structures such as the reorder buffer and the branch history table by detecting errors in the array rows, abstracting the fault pattern to investigate variable-granularity fault tolerance, and the investigation of metrics used to evaluate performance.

There has also been work that targets defect tolerance specifically at the control logic of processors [37, 2] by employing the flexibility of field-programmable logic to protect the control of the microprocessor from faults.

As an alternative to coarse-grain reconfigurable hardware, a large number of works exploit FPGAs to build reliable systems. For example, NASA [32] and CHREC [26] currently use FPGAs to build reliable systems for Space Computing. In the past years many other research works proposed FPGAs for fault-tolerance [3, 30, 20, 19].

On one hand, core-level redundancy appears to offer limited defect tolerance in the presence of increasing fault rates expected in future technologies. On the other hand, FPGA-based solutions incur reduced performance and high power consumption. Redundancy at the granularity of pipeline stages offers a good balance between defect-tolerance levels and performance [15], however, none of the previous works, such as StageNet and CCA, address the problem of searching for an efficient configuration to tolerate defects. That is exactly the problem treated in this paper.

3.5 Conclusions

In this chapter we gave a detailed definition of the problem that we will approach and defined all the assumptions that we make. We defined the system that we work on as well as the exact problem that we address, and also presented some related research and work.

Proposed Approach

In this chapter we describe in detail the approach that we took to address the problem that we defined in chapter 3. We define the problem domain that we chose to work with in section 4.1, describe some of the specific details that arise when working in this problem domain in section 4.2, we define the heuristic functions in section 4.3 and finally we describe the heuristic search algorithms in section 4.5.

4.1 Choosing the problem domain

While the problem itself has been defined in chapter 3, the domain that the problem resides in is actually a choice. For this particular problem, a number of domains could possibly apply, such as routing, graph coloring, shortest path, constraint satisfaction, perhaps even (integer) linear programming.

We chose to treat this as a **combinatorial problem**, because this allowed for a more intuitive definition of all aspects of the system defined in Figure 3.6 and also provided methods for solving that are flexible both in time and in accuracy demands, an essential option for the system designer.

4.1.1 Combinatorial analysis

As stated in Chapter 3, the problem is finding a (re-)configuration of the multiprocessor array such that some metric of performance is maximized. Before going into detail as to what this metric can be, a way of looking into possible (re-)configuration options has to be developed.

In essence, a configuration of the array is just an arrangement of the available parts into *functional* processors. Arrangements have been discussed in section 2.3. There, we mentioned that the relevant arrangements are **permutations** and **combinations**. This is because reconfiguring an array means **selecting** the parts that will be used to form a new array (combination), and then placing them in a certain **sequence** (permutation). What remains to be defined is the nature of the element that needs to be selected and how to define these sequences, in order to properly apply this combinatorial analysis to the problem.

As mentioned already in section 3.1, a processor needs all of its pipeline stages in order to function properly. In fact, we can state that *a processor needs **exactly one** instance of each pipeline stage*. This statement may seem trivial, but it has a significant impact on the combinatorial approach for this problem.

A multiprocessor array can have N processors, each one with M stages. Naively applying combinatorial analysis would result in investigating arrangements of $N * M$ elements. However, exactly because of the statement above, we can see that there is no

point in examining arrangements that produce a processor without one instance of each stage, or a processor with multiple copies of the same stage. This significantly reduces the amount of arrangements that need to be investigated.

Furthermore, the observation can be made that, because of the statement above, we can individually study arrangements of each *column* (pipeline stage set) in the array, and then placing these arrangements one after the other to form the array, since this would create processors with exactly one instance of each pipeline stage. In other words, the arrangements of each column are **combinatorially independent** from one another. This does not mean that changing the arrangement of one column does not affect the performance of the array as a whole. It only means that arranging one column in a specific way does not interfere with the arrangement of other columns in the array. A column can be individually re-arranged and still produce a valid (but different!) multiprocessor array.

That said, we can now formally define what exactly constitutes an arrangement of a column. Let A be a multiprocessor array of size $N * M$, with a set of defects. As discussed above, in order to create k processors from A , we need to **select** k elements from each column. This is a k -combination, as defined in section 2.3. This is followed by selecting a permutation of these k elements. These two steps define an arrangement of a column, which is a k -permutation. In other words: The set of possible re-configurations of A is equal to the set of all k -permutations of all columns of A .

4.2 Problem specifics

4.2.1 Bottleneck columns

The amount of processors k to be created after defects have been detected is upper bound of course by the amount of processors N that were initially available ($k \leq N$). However, we can further observe that since, as stated in section 3.2, there is no sharing of stages amongst processors, and the fact that each processor needs at least one element from each column, k is also upper bound by the amount of copies of a stage available in the column with the most defects. We call this column a *bottleneck column*. It should also be obvious that there can be more than one bottleneck columns in a multiprocessor array.

Figure 4.1 shows an example 4x5 array with the bottleneck columns marked. It can be seen that even though $N = 4$, k is also bound by the amount of elements in the bottleneck column stage, so $k \leq 2$

4.2.2 Problem size

Having defined the exact arrangement that needs to be investigated per column, and given the total amount of columns in an array, we can now deduce the actual input size of the problem.

As stated in section 2.3, the amount of k -permutations in a set is equal to the total amount of permutations per k -combinations of the same set. We are also interested in the amount of available elements in each column, because the k elements have to be selected from that set. In the case of bottleneck columns, this number is k , but for any other

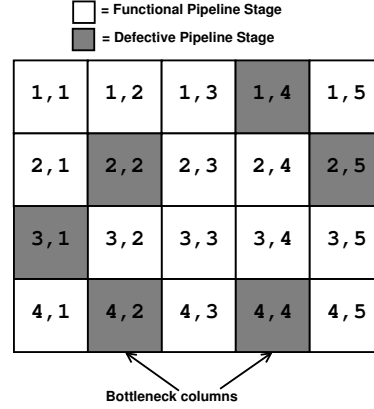


Figure 4.1: A 4x5 defective multiprocessor array, with bottleneck columns marked

column this can be any number n with $k < n \leq N$. So, with n defined for each column by defect detection, we can calculate the total amount of reconfigurations r for a column, as being equal to the amount of permutations times the amount of k -combinations:

$$r = \binom{n}{k} k! = \frac{n!}{k!(n-k)!} k! = \frac{n!}{(n-k)!}$$

However, this is only a part of the total amount of reconfigurations for the entire multiprocessor array. For each possible reconfiguration of a column j , all other possible reconfigurations of all other columns result in different arrays. So, the total amount of reconfigurations R of an array with M columns is:

$$R = \prod_{j=1}^M r_j = \prod_{j=1}^M \frac{n_j!}{(n_j - k)!}$$

4.2.2.1 External constraint impact on size

In the equations above, n was defined by the defect detection module. It is not a free variable but rather a constant that is defined for each column. In this problem, k is the free variable that determines input size. Having defined n , we can now describe the upper bound of k as

$$k_{max} = \min_{j=1}^M n_j$$

i.e. the amount of elements in a bottleneck column, or the maximum amount of working processors that can be created from a defective multiprocessor array.

This statement holds true, in the absence of external constraints. External constraints however complicate things, as they can also be defined as a function of the entire array. That is to say, the information on whether a given reconfiguration of an array violates a constraint or not cannot be extracted from the individual arrangements of each column (i.e. as the array is being created); The array as a whole has to be investigated.

There can be constraints that are restrictive enough such that k cannot obtain its maximum value, and this information cannot become available until an array with k processors has been generated and examined. In other words, given strict enough external constraints, it may not be possible to create an array with k_{max} processors, even if there are enough parts to create them.

This, in effect, forces us to consider all possible arrays with $k = 1, 2 \dots k_{max}$ processors. So, the amount of reconfigurations R that needs to be investigated becomes:

$$R = \sum_{k=1}^{k_{max}} \prod_{j=1}^M \frac{n_j!}{(n_j - k)!}$$

So, even if external constraints reduce the amount of valid reconfigurations, the fact that they can only be applied until after an array has been generated in fact increases the problem size.

The growth rate of R as a function of k is a typical example of **combinatorial explosion** (and is in fact an NP-complete problem since it includes the permutation and combination generation problem [9]), a phenomenon that will greatly impact both the choice of algorithms discussed later in this chapter, as well as their performance shown in chapter 5. It should also be noted that k grows in the “inverse” direction of the variable that we are conceptually working on, namely the amount of defects on a given multiprocessor array; in fact, they complement each other per column, to N . This leads to a “peculiar” situation where more defects within a multiprocessor array lead to a smaller (easier) problem. Seen however from a more abstract point of view, this becomes intuitive easily: The more the defects in a multiprocessor array, the less the choices available to investigate, leading to a smaller input size for the problem, and vice versa.

It is also perhaps worth noting that the special case where $k = N$ means that there can be as many processors as there were there to begin with. In other words, $k = N$ indicates that there are no defects in the multiprocessor array, so from the defect tolerance point of view in this thesis, this case is of no interest.

4.3 Heuristic functions

For this particular problem, what needs to be maximized is the performance of the multiprocessor array as a whole. From the definition so far, we can formulate it as an **optimization** problem as described in section 2.3, which is finding the reconfiguration X of the multiprocessor array that will maximize the performance function $f(X)$.

The obvious question here is: What is $f()$? As described in section 2.4, we do not always know this function, so we need to use approximations called **heuristic functions**, $h(X)$. This problem is one of those cases since we do not know the details of the interconnect implementation and how these would affect the specific architecture of each processor in a multiprocessor array, we cannot define $f(X)$ and instead use $h(X)$ functions.

For this thesis, we define 3 different heuristic functions, each taking into account increasing degrees of prior knowledge about the interconnect implementation, processor

architecture and interaction between the two.

4.3.1 Manhattan distance

The first and most intuitive way to measure how much a processor has been affected compared to its initial configuration is to measure how far each stage is from one another in the new configuration. Euclidean distance however is not of much use in this case, since the two-dimensional plane is not continuous but discrete as we defined in section 3.2. Measuring the index units of the length l of each processor in the array gives a more accurate picture of how much that processor was changed. This metric of distance by array index units is called **Manhattan distance**. Naturally, we would like for the length l of a processor to be **minimized**, since a smaller distance would imply less impact on either latency, power, throughput, etc.

If a processor was not reconfigured at all and is at the same state as before defects were detected, then $l = M$. This means that the “perfect” state of a processor is offset by M . We can subtract M from the length of a processor in order to set the minimum value to 0, or, equivalently, we can ignore the horizontal index units when counting the length l . This, in effect, means that we only measure the **vertical** Manhattan distance of a processor; The horizontal distance will always be equal to M .

With these definitions the function $h_{MD}(x)$ (MD stands for Manhattan Distance) of a reconfiguration x for a **single processor** can be properly defined as:

$$h_{MD}(x) = \sum_{j=1}^{M-1} |x_j - x_{j+1}|$$

Here we assume that x holds the **vertical** (row) indices of each element of the processor in the original array A .

Having defined the function for reconfigurations of a single processor, the function for the entire array $h_{MD}(X)$ is simply the sum of all processors’ individual functions:

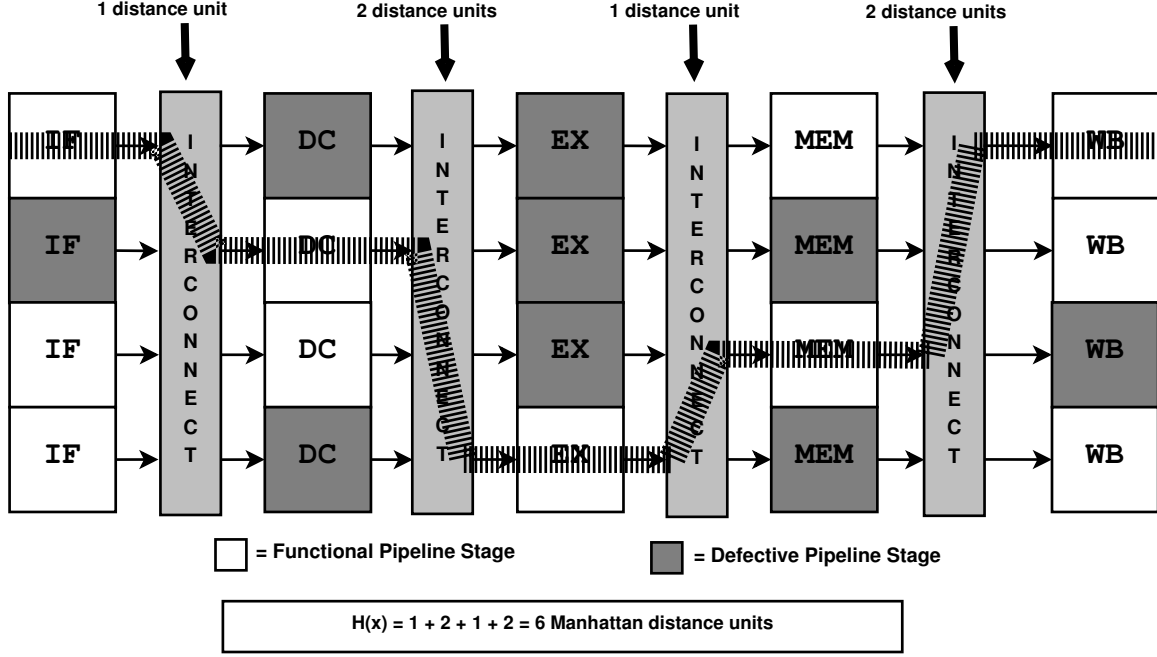
$$h_{MD}(X) = \sum_{i=1}^k h_{MD}(x_i) = \sum_{i=1}^N \sum_{j=1}^{M-1} |x_{i,j} - x_{i,j+1}|$$

So, the question of the combinatorial problem of **optimization** when using this function becomes a problem of **minimizing** function $h_{MD}(X)$. In Figure 4.2 we show a visual representation of the formal definition for this heuristic.

There is one issue however with this function as it is defined right now: As mentioned in section 4.2, due to the effect of external constraints all configurations with $k = 1, 2, \dots, k_{max}$ must be examined. This leads to an undesirable behaviour when using this heuristic, namely that an array with more processors will have a worse or equal evaluation than an array with less. In other words, the heuristic favours the construction of arrays with as few processors as possible. The proof is relatively simple:

Proof. From the definition of $h_{MD}(x)$ we can easily see that:

$$h_{MD}(x) \geq 0$$

Figure 4.2: An example processor x and it's evaluation $h_{MD}(x)$

Also:

$$h_{MD}(X) = \sum_{i=1}^k h_{MD}(x_i)$$

Given two candidate configurations X_1 and X_2 with k_1 and k_2 working processors respectively, with $k_1 < k_2$, we can define:

$$h_{MD}(X_1) = \sum_{i=1}^{k_1} h_{MD}(x_{1i}), \quad h_{MD}(X_2) = \sum_{i=1}^{k_2} h_{MD}(x_{2i})$$

Since $h_{MD}(x) \geq 0$ and $k_1 < k_2$ it follows that:

$$h_{MD}(X_1) \leq h_{MD}(X_2)$$

And due to the fact that this is a minimization problem:

$$h_{MD}(X) = \min(h_{MD}(X_1), h_{MD}(X_2)) \Rightarrow k = k_1$$

□

To accommodate for this undesirable behaviour, an extra rule is added in this heuristic to always choose amongst the configurations X that have the largest value for k .

4.3.2 Throughput loss

The previous heuristic function used almost no knowledge of either interconnect implementation or computer architecture, and instead described a simple way of measuring the length of a processor. While this may be a good function to attempt to minimize, we can incorporate more knowledge or estimates about the internals of this system in a heuristic function.

We define the heuristic function h_{TL} (TL stands for Throughput Loss), which takes into account a simple fact: Increasing the delay of the critical path in a pipeline by a factor of c , results in clock frequency decreasing by a factor of $1/c$, therefore directly reducing throughput by a factor of $1/c$ [17]. This could be avoided by distorting the pipeline balancing (moving elements of the pipeline from one stage to another) to change the delay of each stage, but such changes would need to be performed dynamically depending on defect distribution, and this is beyond the scope of this thesis.

This still leaves the open issue of how to estimate an increase in the critical path delay. Given only the above fact, one estimation that we can make is that increasing the length of a pipeline stage by c is essentially adding c times the delay to the critical path. This may not be the case depending on interconnect implementation, and this is why this remains an estimate and the function is just a heuristic.

The main difference between this heuristic and the Manhattan distance one is that clock frequency is affected by the **maximum** increase in delay. In other words, only one delay increase will affect the estimated throughput of a processor, instead of the accumulating effect of the Manhattan distance heuristic. This becomes more obvious by formally defining the heuristic $h_{TL}(x)$ for a reconfiguration x of a processor, with the same assumption for x as before, namely that it holds the vertical (row) indices of the pipeline stage parts that the processor is made of. The final estimated throughput is expressed as a percentage of the original throughput:

$$h_{TL}(x) = \frac{100}{\max_{j=1}^{M-1} |x_j - x_{j+1}|}$$

As with the Manhattan distance heuristic, the heuristic value of the entire array $h_{TL}(X)$ comes from summing the heuristic values of the individual processors:

$$h_{TL}(X) = \sum_{i=1}^k h_{TL}(x_i) = \sum_{i=1}^N \frac{100}{\max_{j=1}^{M-1} |x_{i,j} - x_{i,j+1}|}$$

In Figure 4.3 we show a visual representation of the formal definition for this heuristic.

It should be obvious that since the metric here is throughput, the optimization goal is to **maximize** function $h_{TL}(X)$. It is also relatively easy to prove that this heuristic function $h_{TL}(X)$ needs no corrective rule like the Manhattan distance heuristic $h_{MD}(X)$, and that it will always choose the maximum value for k that is available:

Proof. Let's assume that there is a X_1 with $k = k_1$ and a X_2 with $k = k_2$ such that:

$$k_1 < k_2$$

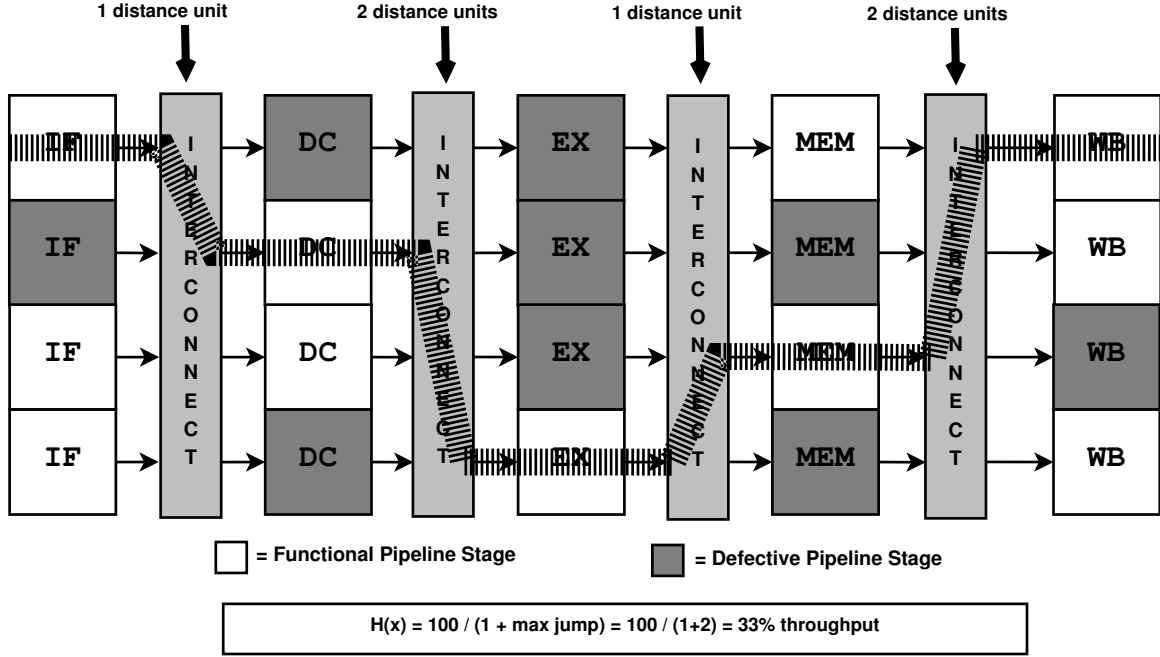


Figure 4.3: An example processor x and its evaluation $h_{TL}(x)$

and:

$$h_{TL}(X_1) > h_{TL}(X_2)$$

where:

$$h_{TL}(X_1) = \sum_{i=1}^{k_1} h_{TL}(x_{1_i}), \quad h_{TL}(X_2) = \sum_{i=1}^{k_2} h_{TL}(x_{2_i})$$

This would imply, since this is a maximization problem, that:

$$h_{TL}(X) = \max(h_{TL}(X_1), h_{TL}(X_2)) = h_{TL}(X_1)$$

In other words, the assumption is that there can be a case where the heuristic will choose a configuration X_1 with less working processors (k_1) given that there exists a different X_2 with more (k_2). We will prove this assumption wrong.

By definition:

$$h_{TL}(X) = \sum_{i=1}^k h_{TL}(x_i)$$

And also, by definition:

$$h_{TL}(x) \geq 0$$

Now, given the two different configurations X_1 and X_2 with $k = k_1$ and $k = k_2$ respectively, we can see that because of the two equations above, we will get:

$$h_{TL}(X_1) \leq h_{TL}(X_2)$$

$$\Rightarrow h_{TL}(X) = \max(h_{TL}(X_1), h_{TL}(X_2)) = h_{TL}(X_2)$$

$$\Rightarrow k = k_2$$

□

Simply put, this means that even the worst possible configuration for a processor will add to the total throughput of the array. If there is a configuration with less processors that is efficient, adding extra processors to it, if it is possible, will only result in better overall performance (according to this heuristic function). As we will show in the evaluation section in chapter 5, given a slow enough processor, this extra performance gain from such a slow processor could take a longer amount of time until it becomes noticeable, but will asymptotically add to overall performance given enough time.

4.3.3 Bubble stage impact

Usually, a computer architect would avoid increasing the critical path delay. Instead they opt for pipelining those critical parts of the operation, splitting them up into smaller pipelined parts. This is only possible, however, if the hardware part that defines the critical path delay can actually be split into smaller parts.

The same could be applied to the interconnect, if indeed it does increase critical path delay and barring pipeline balancing distortion (as discussed above). We could assume for example that the interconnect is pipelined at every row index of the array, in effect adding “bubble” pipeline stages in the processor for every row that the processor has to “travel” vertically to get to the next actual pipeline stage. We define the heuristic function $h_{BI}(x)$ (BI stands for Bubble Impact) as the estimation function of the impact that these bubble stages have on the performance of a processor x . In Figure 4.4 we show what the example reconfigured processor we used in the previous heuristic functions and the interconnect would be like in the case of a pipelined interconnect.

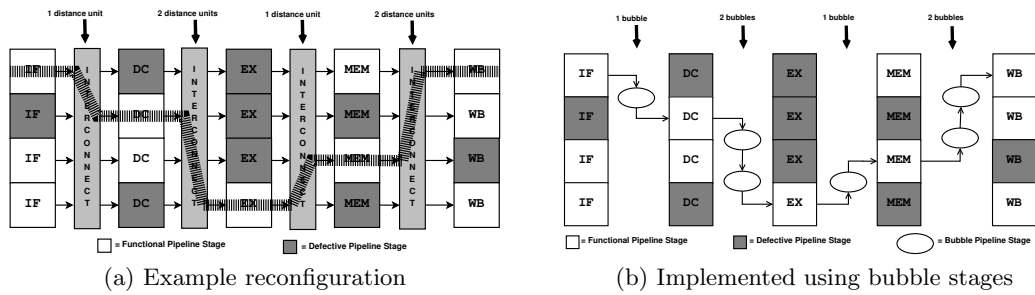


Figure 4.4: Example reconfiguration & respective pipelined interconnect implementation

Pipelining the interconnect instead of increasing the critical path delay has also the benefit of allowing all processors to keep the same clock frequency, instead of having to clock each processor individually according to their respective critical path delays. This is also important, since some multiprocessor array architectures do not allow for multiple clock frequencies amongst their processors.

Before we can properly define $h_{BI}(x)$ however, we need an actual estimation of the impact that bubble stages have. This is not something that can be easily estimated, like the effect of delay on clock frequency. The specifics of the processor architecture have a lot to do with how they are affected by inserting empty pipeline stages between the actual ones.

Because of this, we had to investigate bubble stages on an actual architecture. For that, we used **CoWare's Processor Designer**¹ set of tools, which allow for an easy way of defining an architecture, and automatically build a C compiler as well as a **cycle-accurate** simulator for the architecture that was defined. This way we could easily define an architecture including bubble stages, and simulate the execution of programs on it to measure the impact of these bubble stages.

4.3.3.1 Simulating processors in Processor Designer[®]

First of all, we needed to define the baseline pipeline that would be injected with bubble stages. We used the typical 5-stage pipeline defined in Figure 2.1. Processor Designer[®] provides a basic implementation of such a pipeline.

Creating functional processors with bubble stages

Injecting bubble stages as a problem can be broken down into the problems of injecting bubble stages between each pair of actual pipeline stages. In a 5-stage pipeline, specifically the one we defined in figure 2.1, there are 4 points of injection: Bubbles between IF & DC, DC & EX, EX & MEM and MEM & WB stages.

As such, we need to define 4 sets of rules that define what architectural changes need to be made in each of the 4 cases of bubble injection in order to maintain proper functionality within a pipeline.

First off, we can make one crucial observation: Injecting bubbles between two stages, A & B, is in effect an extension of pipeline stage B. Once an instruction has passed stage A and is in the bubbles before B, the operation that was assigned to pipeline stage A has already been completed. The instruction is on its way to pipeline stage B, so the bubbles between A & B only delay the completion of the operation assigned to stage B.

Using that observation, and by studying the operation of this processor and its hazards described in section 2.1 and in more detail in [17], we can deduce the following sets of rules and changes for injecting bubble stages anywhere in the pipeline:

- **IF/DC bubbles**

Injecting bubbles here means delaying the DC stage of the pipeline. The IF stage has already completed the fetch operation. This means the pipeline runs the risk of having instructions fetched that should not have been fetched (Control Hazard), and also of operations advancing when there is stalling (Structural Hazard). In order to maintain functionality, the following measures need to be taken:

¹CoWare is now part of Synopsys, Inc.

- Stall the fetching operation if the bubbles ahead and the DC stage have been stalled (in order to avoid collision of instructions). This is essentially an extension of the stall detected ahead.
- Whenever there is a branch that is resolved as taken, all bubbles between IF and DC need to be flushed, since they are carrying invalid instructions.

- **DC/EX bubbles**

Injecting bubbles here means delaying the EX stage of the pipeline. The DC stage has already completed decoding the instruction. This means that the pipeline has the same Control & Structural Hazards as before, but now the information on what to stall and when is available. In order to maintain functionality, the following measures need to be taken:

- Add a **global register** to the architecture, called BPEN. This register is a boolean register. It is set whenever there is a branch pending in the pipeline that has not been resolved yet as taken or non-taken.
- Set BPEN by the DC stage when decoding a conditional branch (since unconditional branches are resolved at the DC stage).
- When BPEN is set, the fetch operation at the IF stage is blocked, and all FE/DC bubbles are also stalled.
- When the branch operation has been resolved, clear the BPEN register.

- **EX/MEM bubbles**

Injecting bubble stages here means delaying the MEM stage of the pipeline. The EX stage has already completed executing the instruction. This means that there could potentially be Read-After-Write (RAW) dependencies that can only be resolved by the data in the bubble stages. It also means that the MEM stage is delayed, and RAW dependencies with the MEM stage of the pipeline cannot be resolved and have to be stalled until the offending instruction clears the MEM stage. A RAW dependency at the MEM stage means a **load** instruction - load writes from the memory to a register, another instruction reads that register. In order to maintain functionality, the following measures need to be taken:

- In order to address the non-load RAW hazards, add forward wires from all EX/MEM bubble stages to the DC and EX stages.
- To address load RAW hazards, add an extra **pipeline register** named LDR.
- When a load instruction is decoded, the address of the register to be written by load is written on LDR.
- When checking forwarded values, LDR is also checked. If a match is found on a forwarded LDR, the instruction **stalls along with the rest of the pipeline before it**.
- When the load instruction clears the MEM stage, LDR is set to address 0 (corresponding to register R[0] that always holds the value 0). This allows the stages that check forwarded LDR values to come out of stalling. The

instruction that needed that value can now retrieve it either from regular forwarding or from Write-Back, depending on where it is in the pipeline.

- **MEM/WB bubbles**

Injecting bubbles here means delaying the WB stage of the pipeline. All that is needed in order to maintain functionality is to simply add forwarding wires from every MEM/WB bubble stage to the DC, EX and MEM stages. This clears the only hazard that bubbles introduce here, namely RAW Data Hazards, when an instruction is reading from a register that has not yet been written back due to the delayed WB stage that the writing instruction has still not reached.

The above set of rules contains expressions like “for every bubble stage” or “to all bubble stages” etc. This implies that the action is to be performed on a variable number of bubbles. The number is a variable because we want to implement different processors with different bubble configurations, to measure their impact.

It would be very convenient if the language that Processor Designer[®] uses to describe an architecture was a **reflective** programming language (i.e. a language that can facilitate modification of its own code at runtime). That way we could easily describe an architecture that would apply these rules to all of its bubble stages without having to change the description itself aside from redefining the amount of bubble stages at each injection point.

However, this is not the case. The Processor Designer[®] architecture description language does not support reflection. This means that every time we want to change the amount of bubbles at each injection point, we would have to re-write the parts of the architecture description that are relevant to the rules defined above.

In order to avoid this manual re-writing of code that is actually a pattern based on some parameters, we apply a simple meta-programming technique: A **script** was implemented in the Python programming language that takes as input the amount of bubbles at each injection point, and applies the rules defined above wherever they apply in the processor description. Having such a script allows for a much simpler experimental setup to evaluate the impact of bubble stages. We simply set the parameters, run the script and then compile the new processor.

Simulating bubble-injected processors & evaluating bubble impact

After having found a way to create a functional processor with any amount of bubble stages injected, we need to consider exactly what to simulate and how to evaluate the impact that these bubble stages have on the processor’s performance.

As mentioned before, Processor Designer[®] provides us with a simulator for the processor model that was described. This simulator is cycle-accurate, and that is because a real-time accuracy simulation would require specific parameters about the hardware on which the processor model would be implemented. However, for the needs of a heuristic function, cycle-accurate simulation is sufficient.

It is obvious from the rules defined for bubble injection that different programs would be affected by bubbles in different ways. For example, based on the rules for EX/MEM bubbles we can see that a program with many load-RAW Data Hazards would be stalled

much more than a program without as many. For this reason, we decided to write and simulate the execution of **three micro-benchmarks**: Each one designed to stress the different parts that these rules have an impact on, namely the ratio of taken and non-taken branches, and the amount of load-RAW Data Hazards.

- **“Normal”**: This micro-benchmark has the expected ratio of taken to non-taken branches (the rule of thumb is that 1 out of 5 branches is non-taken [17]). We define the amount of load-RAW dependencies in this benchmark as the nominal amount, which is about 1 out of every 3 instructions is depending on a preceding load instruction.
- **“RAW”**: This micro-benchmark has an increased amount of load-RAW Hazards (double the nominal amount defined above, so 2 out of every 3 instructions induce load-RAW hazards), and a normal ratio of taken to non-taken branches.
- **“Non-taken”**: This micro-benchmark has an increased amount of non-taken branches compared to the taken branches (about half of the branches are taken, the other half not taken), and a the nominal amount (1 to 3) of load-RAW Data Hazards.

In Figure 4.5 the instruction mix of each micro-benchmark is shown, as generated by the compiler that Processor Designer[®] generated. It is worth noting that there was **no re-compilation** of the original source code of the micro-benchmarks when a new pipeline was defined (which means that the compiler did not have to be re-generated either). This was a constraint that we defined. In a possible real implementation the ability to re-compile the source code of the application(s) that run on a system is usually a luxury that cannot be afforded.

Since our simulations can be cycle-accurate, we measure the cycles that each micro-benchmark takes to complete on processors with varying amounts of bubble stages injected. To that end, we used the following injection patterns: We assumed an 8x5 multiprocessor array, which means that the maximum vertical distance from one stage to the next is 7 steps (i.e. the worst case scenario where we need to go from the first row with index 1 to the last row with index 8 to get the next block or vice-versa, $|1 - 8| = 7$ vertical steps). This means that there would be at most 7 bubble stages between regular stages. So we created processors with 0 to 7 bubble stages at each injection point independently. This allowed us to independently see how much impact a bubble has at each injection point.

In Figure 4.6 we can see the results of this simulation. The x-axis is the amount of bubbles at an injection point, and the y-axis shows the cycles that were needed to complete the micro-benchmark. With these evaluations, we can deduce some basic characteristics of the impact that bubble stages have:

- MEM/WB bubbles have no impact at all regardless of benchmark. This is obviously due to the fact that since there are forward wires in the bubbles prior to the WB stage, and since all the WB stage does by definition (see section 2.1) is forward values to the DC stage, delaying the WB stage does not induce any stalling conditions. This can also be seen from the set of rules for injecting bubbles.

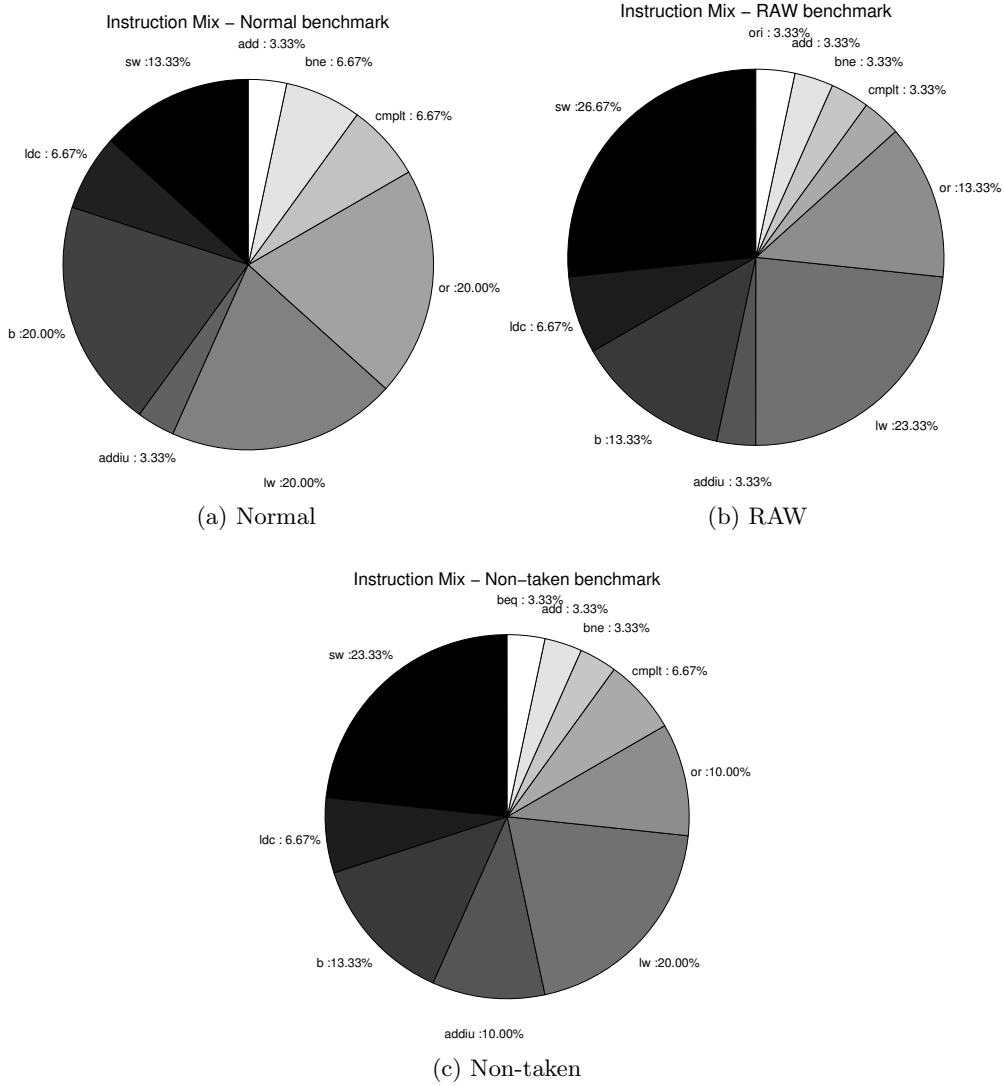


Figure 4.5: Instruction mix of all micro-benchmarks

- IF/DC bubbles and DC/EX bubbles have the same or almost the same impact per bubble inserted for all benchmarks. As expected from the rules defined above, in the case of non-taken branches, DC/EX bubbles have a slightly heavier impact than IF/DC bubbles.
- EX/MEM bubbles seem to have the largest impact on all three benchmarks. This is to be expected since as defined in the rules above, EX/MEM bubbles affect both non-taken branch situations as well as load-RAW hazards. Also expected is the fact that in the case where load-RAW hazards are stressed, EX/MEM bubbles have a steeper impact than in the other benchmarks.
- The impact of bubbles in all injection points in all benchmarks seems to be linear.

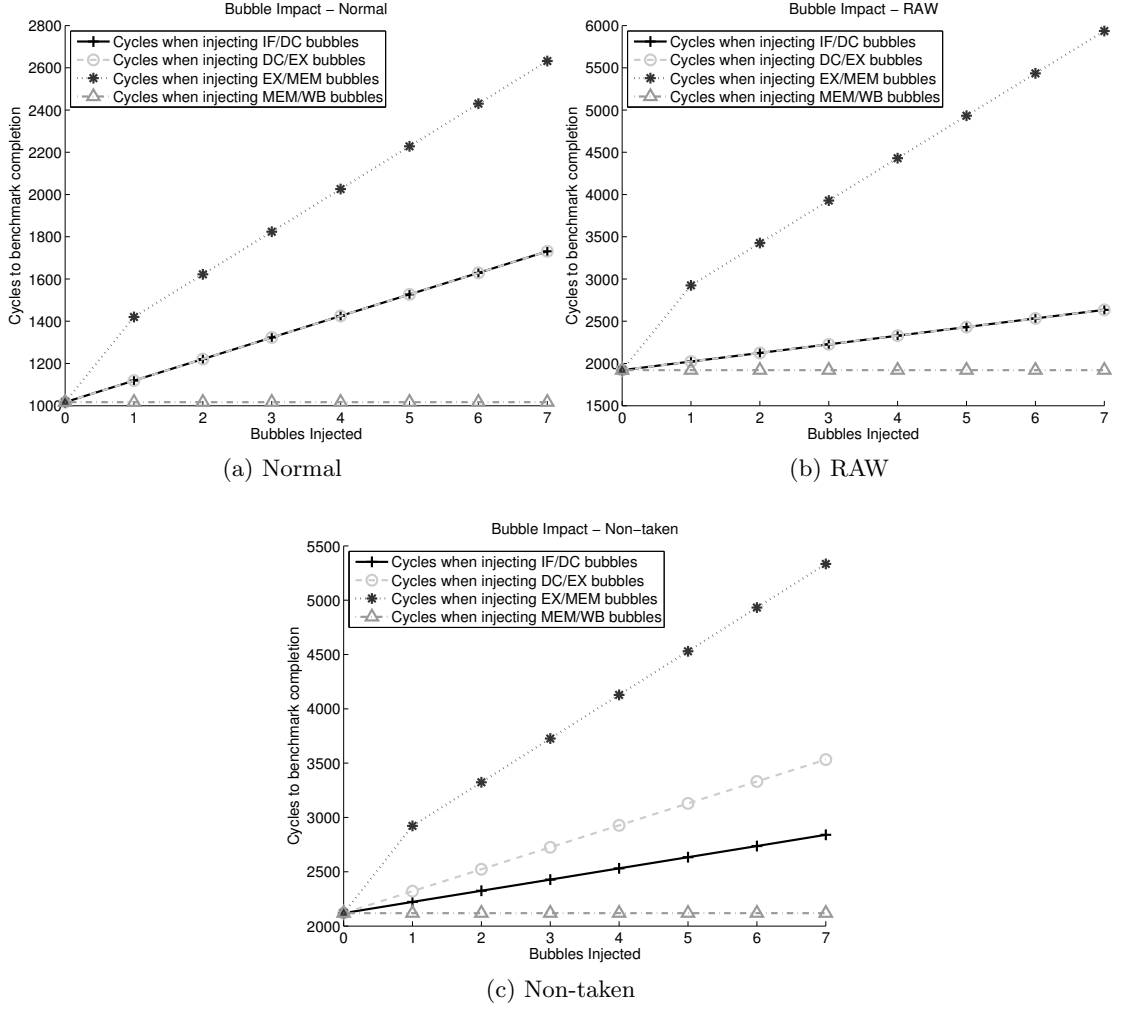


Figure 4.6: Bubble impact evaluation for all micro-benchmarks

The steepness of the line is different for each benchmark but not too different; We can deduce a steepness for each injection point that will apply well enough for a heuristic to all situations.

These deductions give us a good idea of how to define the approximation that is the heuristic function $h_{BI}(x)$ of a processor x . We can, after studying the evaluations above, deduce that the impact of each bubble in a specific injection point increases the cycles that it takes to complete the benchmark by a fixed percentage. After studying the steepness of the plots provided in Figure 4.6, we can define those percentages as such (Figure 4.7):

Of course, a processor may not (and usually does not) necessarily have only one point where bubbles are injected. This heuristic is accurate in the case that each injection point impacts the overall performance independently from the other injection points. As we will show later in chapter 5, this proves to be an accurate assumption.

IF/DC	10%
DC/EX	10%
EX/MEM	20%
MEM/WB	0%

Figure 4.7: Approximation of cycle impact per bubble at each injection point

It should go without saying of course that in the case of an array of application specific processors, such an evaluation should be done with the actual code of the application that will be executed on the array. The more accurate the evaluation of the bubble impact, the more accurate the heuristic function will be.

4.3.3.2 Formal definition of the Bubble Impact heuristic

Having evaluated the actual impact of bubble stages injected in a pipeline, we can now formally define the heuristic function $h_{BI}(x)$ of a processor x .

Given a processor x with bubble stages injected, and an array w that holds the values displayed in Figure 4.7, we can evaluate the amount of cycles that a program will need to complete as a percentage of the ideal configuration where there have been no injections of bubble stages. This can be expressed as a percentage of the original processor speed, in order to formulate this as a **maximization** problem, to avoid corrective rules as with $h_{MD}(X)$.

The heuristic function $h_{BI}(x)$ for a processor x can be defined as follows:

Let $w = [0.1, 0.1, 0.2, 0]$ as defined in Figure 4.7. Then:

$$h_{BI}(x) = \frac{100}{1 + \sum_{j=1}^{M-1} w_j(|x_j - x_{j+1}|)}$$

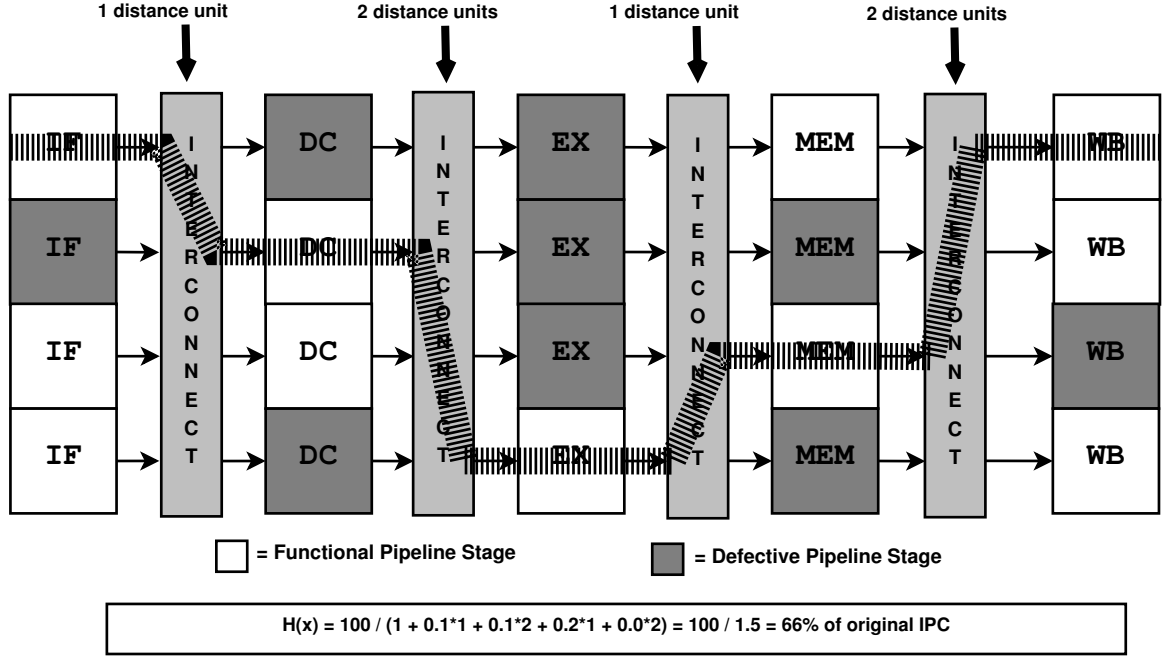
It is obvious for this case that we can apply this function only to arrays with $M \leq 5$ since we used the 5-stage pipeline to extract the approximation of w .

From the formula of the formal definition we can see that the *units* in which the heuristic expresses processor performance is the the reciprocal of the amount of cycles (expressed as a percentage of the ideal processor) needed to complete a known amount of instructions. This, in effect, is a percentage estimation of the expected **Instructions Per Cycle** (IPC) metric. We will use this observation later on in chapter 5 when we compare performance of different arrays. In Figure 4.8 we show a visual representation of the formal definition for this heuristic.

As mentioned already, this function describes the speed of a processor x , so we are looking to maximize this function per processor. The function $h_{BI}(X)$ for a configuration X of the entire array is defined again as the sum of each processor's heuristic value, and hence also needs to be maximized.

$$h_{BI}(X) = \sum_{i=1}^k h_{BI}(x_i) = \sum_{i=1}^k \frac{100}{1 + \sum_{j=1}^{M-1} w_j(|x_{i,j} - x_{i,j+1}|)}$$

The proof that this function will always select the configuration with the largest value for k available is the same as the proof for $h_{TL}(X)$.

Figure 4.8: An example processor x and its evaluation $h_{BI}(x)$

4.4 External constraints

The system also needs to take as an input a set of external constraints, as shown previously in Figure 3.6. These constraints can be any set of rules that are defined over an entire array or even just parts of it. The implementation of the heuristic search algorithms makes no assumptions as to the definitions of these constraints, so they must take the most general case possible only into account, which is the case that a constraint C is defined over the entire multiprocessor array A .

As such, we can view these constraints as boolean functions of A and X . If a configuration X of an array A violates a constraint C then the function will return a false value, otherwise it returns true. So, from now on, constraints will be viewed as functions denoted as $C(A, X)$.

As a proof of concept for the design of the algorithms, we decided to implement a simple constraint. The constraint can be informally described as follows:

“In any given junction point in the interconnect where wires need to connect two stages of a processor, at most L wires can be utilized.”

This means that when an algorithm creates a configuration X of A , it must create it in such a way that there is no vertical overlapping of more than L processor paths. Figure 4.9 shows two configurations of a 4×5 array with $L = 1$ (only 1 vertical wire per junction point), one that violates the constraint and one that does not.

For the evaluation of the work in this thesis (as presented in chapter 5), we used $L = 1$. If the actual implementation of the interconnect is indeed pipelined, the L parameter can

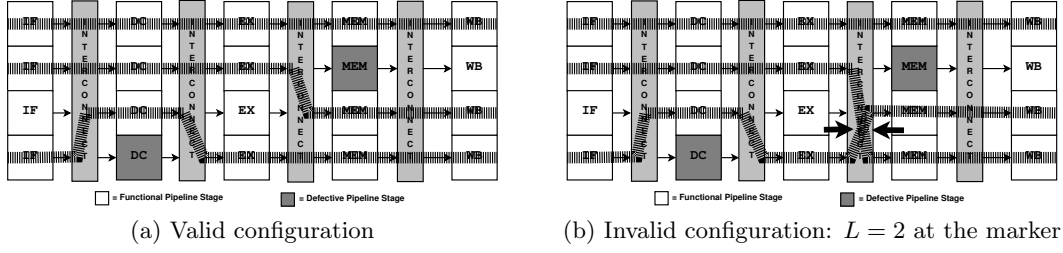


Figure 4.9: Comparative example of two configurations using $C_L(X, A)$ with $L = 1$

be explored to find a value that satisfies area demands of the implementation without making $C_L(X, A)$ too strict.

4.5 Heuristic search algorithms

Having defined the heuristic functions and also described how constraints work, we can now implement algorithms to put these heuristics to use. As discussed in section 4.1, the flexibility to choose between speed and solution quality is valuable, so the algorithms have been chosen and implemented accordingly.

4.5.1 Greedy search

The first approach to the problem was to implement a greedy [7] algorithm, that would be very fast but could also produce decent solutions. This algorithm does not take the complete state formulation approach to the problem. Instead, it builds the solution iteratively, each time taking the best step first. A step means to assign one stage to a processor. So the algorithm, at each step will attempt to assign to the current processor being built, the closest available stage. This is essentially trying to minimize $h_{MD}(x)$, the only function that can be defined at this step level instead of the entire processor.

The process begins by first detecting a bottleneck column. This is because a bottleneck column will have all its parts assigned to a processor, so it is an easy starting point. From there, it moves in two directions, right towards the end of the array and left towards the start. Starting from the bottleneck column, each processor is assigned the closest available part from the column next to the current one before moving one column in both directions.

Naturally there are cases where the same part is the closest available part for more than one processor. In those cases, a First-Come-First-Served policy is applied, meaning that, once a part has been assigned to a processor, it appears as unavailable to the next processor(s) that will search for available parts. Figure 4.10 shows the pseudo-code description of this algorithm which we simply call *Greedy*.

It is obvious that when there are many such contention spots, the optimality of the constructed solution suffers. In the evaluation part of this thesis (chapter 5) the exact quality of the solutions provided by this greedy method is measured. In any case, as this algorithm does not approach the problem with the complete state formulation method,

```

GREEDY.
Input: A (defect array), N, M (array dimensions),  $k_{max}$ 
Returns: X (configuration of A)
  {Find first bottleneck}
   $b \leftarrow M$ 
  for  $j = 1 \rightarrow M$  do
    if  $A[j].workingElements() < b$  then
       $bottleneck \leftarrow j$ 
       $b \leftarrow A[j].workingElements()$ 
    end if
  end for { $bottleneck$  now has the index of the first bottleneck column}
Ensure:  $b \leq k_{max}$  {We can't create more than  $k_{max}$  processors}
  for  $i = 1 \rightarrow k_{max}$  do
    {Assign elements to each processor}
    for  $j = bottleneck \rightarrow M - 1$  do
      {From bottleneck towards the end}
       $X[i][j] \leftarrow closestWorkingElement(A[j + 1])$ 
    end for
    for  $j = bottleneck \rightarrow 2$  do
      {From bottleneck towards the beginning}
       $X[i][j] \leftarrow closestWorkingElement(A[j - 1])$ 
    end for
  end for
  if  $C(X) = True$  then
    return  $X$ 
  else
    return Failure
  end if

```

Figure 4.10: Greedy search algorithm

it is a very fast algorithm since it does not depend on the amount of states that can be generated. Its complexity depends simply on the size of the input array, which is a small input compared to even a fraction of the state space size, as described in section 4.2.

It should also be noted that because *Greedy* works iteratively, it cannot take into account any external constraints that may be defined over the entire array, such as $C_L(X, A)$. This means that *Greedy* will provide a complete solution only after it has finished, and if that solution does not satisfy external constraints, the algorithm will return failure.

4.5.2 Exhaustive search

Having implemented a greedy algorithm, we needed a way to be able to measure just how accurate it was, and this would also apply to any other algorithm that did not

guarantee optimality. Since we have no pre-defined objective function to evaluate a given configuration of a multiprocessor array, we would have to compare the heuristic evaluations of the configurations provided by algorithms. This means that the entire state space would have to be evaluated, to find the best state within it, and then compare the heuristic evaluation of that state with that of the state provided by a search algorithm, in order to measure accuracy.

To that end, we implemented an exhaustive search algorithm that evaluates all states of the state space. A complete state in this problem is a configuration X of the array, and its evaluation is its heuristic value, $h(X)$. An exhaustive search would have to iterate through all the possible configurations and select the best one.

In order to achieve that, we use the combination of algorithms described in Figures 2.4 and 2.5, since as discussed in section 4.1, a configuration is simply the set of a k -permutation for each column. Using these algorithms we can generate all possible configurations of the array in **lexicographical** order. Specifically, for each combination there is a set of lexicographically ordered permutations associated with it, and all combinations are also lexicographically ordered.

As discussed in section 4.2 columns are combinationally independent from one another. This means that the state generation can be applied individually per column of the array. In other words, there are M different sets of states, called *column states*, which, when combined, form the complete state X . Figure 4.11 shows how a configuration X of a 4x4 array A with $k = 2$ is represented as a complete state, using column states.

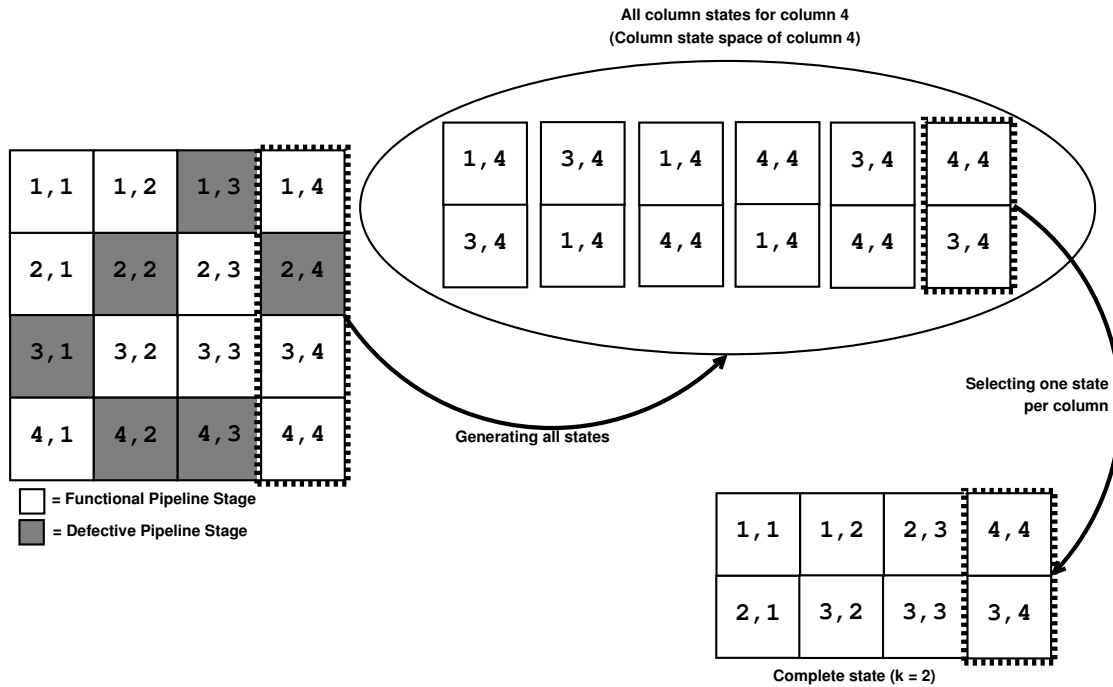


Figure 4.11: Representation of complete states composed of column states

From that figure we can see visually how each column is independent in terms of its

own state space: We can re-arrange the elements within a column and still produce valid (barring external constraints) configurations.

Figure 4.12 shows the pseudo-code description of the exhaustive search algorithm, which we call *BruteForce*. The algorithm generates in order all states of each column, before proceeding to the next and resetting the ones before it, much like a counter is resetting previous digits when incrementing the next by one.

BRUTEFORCE.

Input: A (defect array), N, M (array dimensions), k_{max}

Returns: X (configuration of A)

```

{Set each column of X to the first permutation of the first combination}
for  $j = 1 \rightarrow M$  do
     $X[j].resetCombination()$ 
     $X[j].resetPermutation()$ 
end for
 $cbest \leftarrow X$  {Initialise best to current X}
for  $j = 1 \rightarrow M$  do
    {Now, iterate through all states, replacing best when encountering a better one}
    {Re-arrange X column-wise}
    if  $X[j].isLastPermutation() = False$  then
         $X[j].nextPermutation()$ 
    else
        if  $X[j].isLastCombination() = False$  then
             $X[j].nextCombination()$ 
             $X[j].resetPermutation()$ 
        else
            if  $j \neq M$  then
                {If we have completely re-arranged this column and there are no more ar-
                rangements, reset it before moving to the next, unless this is the last one}

                 $X[j].resetCombination()$ 
                 $X[j].resetPermutation()$ 
            end if
        end if
    end if
    if  $C(X) = True$  and  $best(X, cbest) = X$  then
        {Check if this new X satisfies constraints and if it is better than the currently
        stored cbest}
         $cbest \leftarrow X$ 
    end if
end for
return cbest

```

Figure 4.12: BruteForce algorithm

It is obvious from the formulas describing the size of the state space in section 4.2 that this algorithm is not a fast algorithm, because it scales with the state space, which as shown grows at very fast rates.

4.5.2.1 Look-up tables

A careful examination of the way the generation of combinations and permutations works and the way that they are utilized by the *BruteForce* algorithm will reveal one drawback: After generating all the states of one column, and iterating to the next state of the following column, all the states for the first one have to be re-generated. This is work that has already been done.

In order to avoid this, we can use a Look-up Table (LUT) in order to generate and store all the states per column. This way, whenever we iterate to the next state of a column, we do not have to regenerate all the states for the previous columns again. We simply retrieve them from the LUT. The structure of the Look-up table is explained visually in Figure 4.13, based on the representation of a complete state that was given in Figure 4.11.

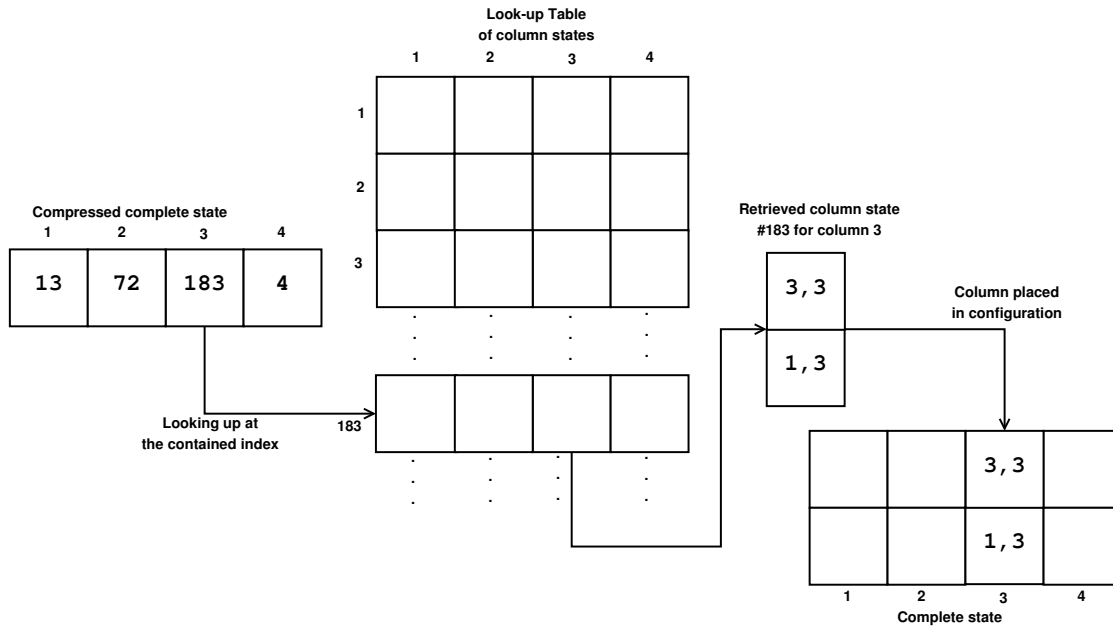


Figure 4.13: Retrieving a column state from the LUT with $k = 2$

With this LUT we can speed up the *BruteForce* algorithm somewhat. An description of this sped up version of *BruteForce* called *FastBruteForce* in pseudo-code can be seen in Figure 4.14.

This method actually provides us with more features that will be essential when implementing other search algorithms:

- It allows the **compression** of the representation of a state from an two-dimensional array to an one-dimensional array that only holds the indexes of each column's

FASTBRUTEFORCE.

Input: A (defect array), N, M (array dimensions), k_{max}

Returns: X (configuration of A)

{Create the LUT containing all possible states per column} {Also initialize X to the first state per columns}

for $j = 1 \rightarrow M$ **do**

$LUT[j] \leftarrow generateAllColumnStates(A, j, k_{max}, N, M)$

$X[j] \leftarrow 1$

end for

$cbest \leftarrow X$

for $j = 1 \rightarrow M$ **do**

for $n = 1 \rightarrow n_j$ **do**

 { n_j is the amount of states per column}

$X[j] \leftarrow n$

$decompress(X, LUT)$

if $C(X) = True$ **and** $best(X, cbest) = X$ **then**

 {Check if this new X satisfies constraints and if it is better than the currently stored cbest}

$cbest \leftarrow X$

end if

end for

end for

return $cbest$

Figure 4.14: FastBruteForce algorithm

state in the LUT. In order to decompress that state, we simply retrieve the column state at the index pointed to by the respective index in the compressed state representation.

- Moving from one state to any other has the same complexity, which is the cost of M look-ups, i.e. the cost of decompressing a single state, instead of having to re-generate all in-between states until we reach the desired state, as would have been the case when using the simple state generation used in *BruteForce*.
- This representation also allows to move to *previous* states, something which was not possible when using the simple state generation algorithms, that only provide the *next* combination or permutation. This is an essential feature for algorithms that traverse the state space in a non-sequential manner like Simulated Annealing, as defined in section 2.4.
- A state representation of this type is also ideal for use with Genetic algorithms (as discussed later but also in section 2.4), where populations are usually described as either bit-streams or as one-dimensional integer arrays. This allows for much simpler breeding and mutation functions.

Of course, using the LUT technique does not have only advantages. The cost to pay for the advantages provided is the memory that is allocated to store all states of each column. This is shown in detail in the evaluation in chapter 5.

4.5.3 Landscape investigation

After having implemented exhaustive search algorithms that order the state space, it is now possible to actually plot out the landscape of the state space as described in section 2.4. We can only do this for input sizes that can be searched exhaustively within a practical time period, but even that should give us a good enough insight on what techniques would have a better chance of providing close to optimal solutions.

The actual shape of the landscape is defined by the evaluation of each state in it. Since we do not have an objective function and we use heuristic functions instead, the landscape is formed by the evaluation of the heuristic. Since we have defined three different heuristics, we will obtain three different landscapes. The input size, as discussed in section 4.2, is a function of k_{max} , which is the maximum amount of working processors one can create out of a given array.

Exhaustive search techniques cease to be practical after $k_{max} \geq 5$ (see also evaluation in chapter 5). For this reason, we plotted out the landscapes for input sizes up until $k_{max} = 4$. Even with only a few input sizes to work with, we can still draw some interesting conclusions from the landscapes that we plot. In Figure 4.15 we can see one of the landscapes that resulted from each of the three different heuristic functions defined, for $k_{max} = 2$ on a 4x5 array. We can see several important features of this landscape that will later help us design more algorithms:

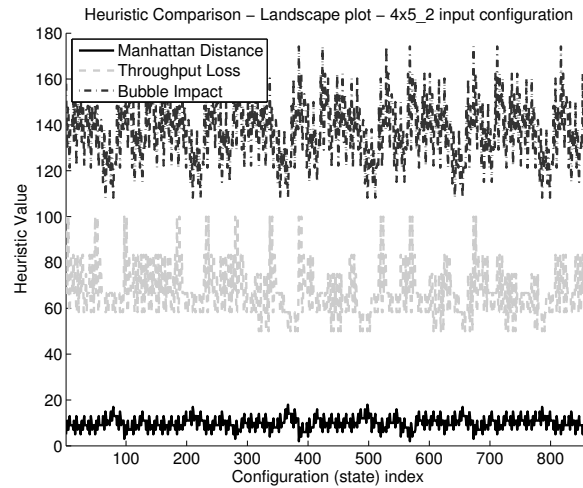


Figure 4.15: Landscape of a 4x5 array with $k_{max} = 2$

- The landscape is full of local extrema, very close to one another. Almost every point on the landscape is actually a local extremum.

- There appears to be some sort of periodicity in the shape of the landscape. Some features and patterns tend to re-appear, and this includes **global** extrema which is very important.
- These characteristics apply to all three landscapes generated by the three heuristic functions. This is not to say that all three heuristics are equivalent. It only means that all three will affect the performance of an algorithm in approximately the same way. The accuracy of the solution still depends on the actual accuracy of the heuristic as well.

4.5.4 Hybrid search using Simulated Annealing

As mentioned already when discussing exhaustive search techniques, their time complexity becomes impractical very quickly as the input size increases. If larger input sizes have to be searched then other techniques must be implemented.

As mentioned in section 2.4, when exhaustive search becomes impractical, one way to work around it is to use local search techniques. These techniques will investigate only a portion of the state space. Their performance will largely depend on the surrounding landscape. However, using Simulated Annealing, the impact of a hard to navigate landscape is lessened.

This has another drawback however: A simple application of Simulated Annealing on such a large state space would result in the quality of the solution quickly decreasing as the state space increases, since Simulated Annealing works as all local search techniques on a specific time limit. If we simply scale the time limit based on input size, we quickly run into the same problem that exhaustive search has, namely the time complexity quickly becoming too big for any practical application.

So, the trade-off here is rather fundamental: We can either sacrifice the quality of the solution and apply a fixed-limit Simulated Annealing search regardless of input size, or we can scale the time limit based on input size to increase the quality of the solution but sacrifice time complexity. In order to work around this trade-off as well as possible, we propose a technique we call *Hybrid* search, combining aspects of Simulated Annealing and exhaustive search. We take advantage of the fact that Simulated Annealing can provide good solutions under a fixed time limit, provided that the state space it searches through is not too big, in order to create a state space that is small enough for exhaustive search.

In order to use this to our advantage, we partition the state space into chunks that have a manageable size, such that Simulated Annealing under a fixed time limit will provide the best state within that part with a good probability. We, then, apply Simulated Annealing to each of these chunks, and from each one we obtain the best state that Simulated Annealing could find. This way we create a new set of states, much smaller than the original state space, with a good probability of containing the globally best state of the underlying state space. We, then, employ an exhaustive search in this new state space, selecting the best state that the exhaustive search finds as the solution that the *Hybrid* algorithm returns.

In Figure 4.16 the pseudo-code description of the algorithm is given. It can be seen that the amount of Simulated Annealing instances and their time limits is a parameter

that can be tuned to give either more accuracy or more speed, the two extremes being either to use only exhaustive search, or to use only Simulated Annealing in the blending of the algorithms.

HYBRID.

Input: A (defect array), N, M (array dimensions), k_{max} ,

S (time limit for Simulated Annealing), T (amount of Simulated Annealing instances)

Returns: X (configuration of A)

for $j = 1 \rightarrow M$ **do**

$LUT[j] \leftarrow generateAllColumnStates(A, j, k_{max}, N, M)$

$X[j] \leftarrow 1$

end for

$cbest \leftarrow X$

for $t = 1 \rightarrow T$ **do**

 {Run an instance of Simulated Annealing in this part of the state space}

 {The tuple $[LUT, t, T]$ defines the chunk of the state space that Simulated Annealing works on}

$X \leftarrow SimulatedAnnealing([LUT, t, T], S)$

$decompress(X, LUT)$

if $C(X) = True$ **and** $best(X, cbest) = X$ **then**

 {Check if this new X satisfies constraints and if it is better than the currently stored $cbest$ }

$cbest \leftarrow X$

end if

end for

return $cbest$

Figure 4.16: Hybrid algorithm

For this implementation of Simulated Annealing, the scheduling function that we used was that of a linear decay that begins from probability = 1 and decays linearly to 0 in S time units (iterations). So the definition of “temperature” as a function of iterations s is:

$$T(s) = -\frac{1}{S}s + 1$$

The reason for this is that because of the extreme variation in the state space landscape, we want to maintain a certain degree of randomness even towards the end. Using exponential decay would result in the Simulated Annealing algorithm performing similarly to a simple Hill Climbing algorithm, i.e. converge quickly and get trapped. By maintaining some randomness, we can have at least a small chance of escaping local extrema even as the time limit is being approached.

To further help the Simulated Annealing algorithm escape from local extrema easier, in those cases that there is no better option, we added the ability for the algorithm to

perform a **random restart** within the allocated state space chunk. This allows the algorithm to visit a wider range of states should it get stuck somewhere.

4.5.5 Partial search

The hybrid approach tried to work around the first observation about the landscape, namely that it is filled with local extrema. We can also take advantage of the second observation, which is that the landscape shows some periodicity. The idea is that applying the *Hybrid* algorithm only to a fraction of the actual state space will give us a faster algorithm. Given a wide enough window there is a very good chance that, because of this apparent periodicity, a global extremum will be found within this window.

So, we introduce a new algorithm called *Partial*, where we apply this technique. First we select a random starting point within the state space, and then expand in both directions until the pre-defined window length is reached, and then apply *Hybrid* within this window. A pseudo-code description of this algorithm is shown in Figure 4.17.

As shown, this algorithm has an extra tunable parameter, which is the width of the window. The wider the window that we use, the slower but potentially more accurate the algorithm will become.

4.5.6 Seeded search

Another approach attempts to improve upon the solution of the *Greedy* algorithm. The idea here is that there could, potentially, be optimal solutions just a few “tweaks” away from the configuration that the *Greedy* algorithm provided. A “tweak” in this problem can be a few swaps between elements in a column, either to a new permutation or to a new combination.

This directly applies to the definition of a complete state that the other algorithms search amongst: Neighbouring states differ from one another just by a few swaps of elements. If we performed a sort of “reverse” look-up, from the configuration X_G that the *Greedy* algorithm provides as a solution, and use that as a pivot to apply a local search, we would essentially be performing exactly this search of “tweaking” on the solution provided by *Greedy*.

A new algorithm, called *Seeded*, first runs the *Greedy* algorithm, performs a “reverse” look-up of the configuration that *Greedy* provides in order to represent it as a compressed state. We need a compressed state since, once we have it, we can use that as the *seeding* point for an application of *Partial* search that uses the compressed X_G as a starting point instead of using a random starting point. Performing a reverse look-up is described by Figure 4.18. Basically finding the compressed representation of a full state given the LUT consists of performing M string-matches of each column in X_G against each element in each column of the LUT. The indices of each match are the compressed representation of X_G .

In Figure 4.19 the *Seeded* algorithm is described in pseudo-code. The parameters of this algorithm are the same as the ones that *Partial* search takes. In the special case that the solution provided by *Greedy* as a seeding point violates external constraints, *Seeded* begins using a random point, which is equivalent to using *Partial* search.

```

PARTIAL.
Input: A (defect array), N, M (array dimensions),  $k_{max}$ ,
S (time limit for Simulated Annealing), T (amount of Simulated Annealing instances),
W (width of window)
Returns: X (configuration of A)
  {Generate the state space}
  for  $j = 1 \rightarrow M$  do
     $LUT[j] \leftarrow generateAllColumnStates(A, j, k_{max}, N, M)$ 
    {Select a random starting point}
     $X[j].randomize()$ 
  end for
  {Now we need to only select those states around the pivot point}
   $cropStates(LUT, X, W)$ 
  {And now we apply the hybrid search}
   $cbest \leftarrow X$ 
  for  $t = 1 \rightarrow T$  do
    {Run an instance of Simulated Annealing in this part of the state space}
    {The tuple  $[LUT, t, T]$  defines the chunk of the state space that Simulated Annealing
    works on}
     $X \leftarrow SimulatedAnnealing([LUT, t, T], S)$ 
     $decompress(X, LUT)$ 
    if  $C(X) = True$  and  $best(X, cbest) = X$  then
      {Check if this new  $X$  satisfies constraints and if it is better than the currently
      stored  $cbest$ }
       $cbest \leftarrow X$ 
    end if
  end for
  return  $cbest$ 

```

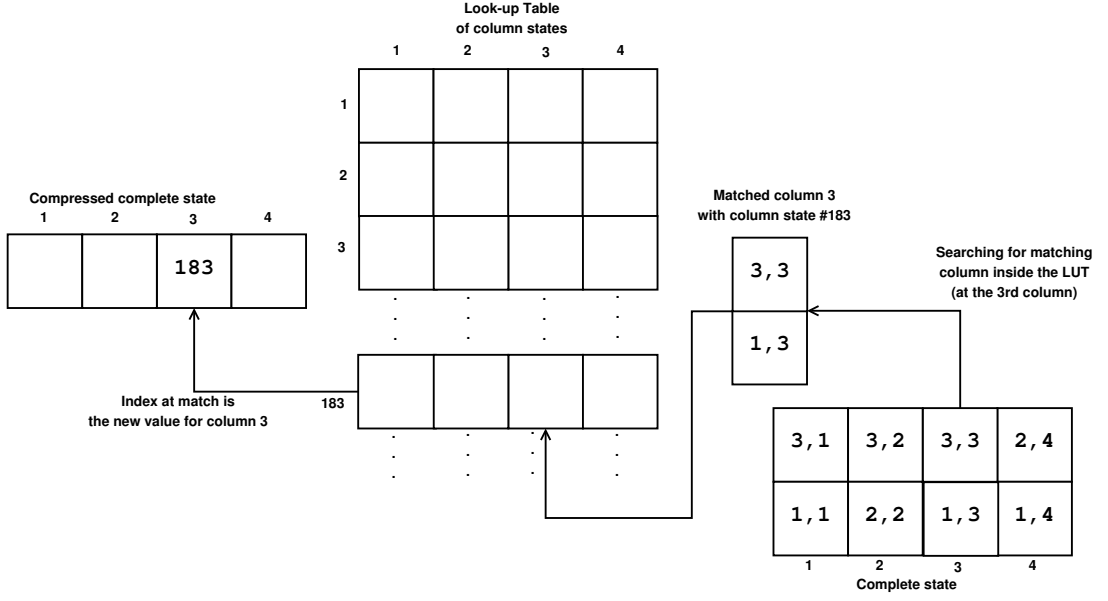
Figure 4.17: Partial algorithm

4.5.7 Genetic search

Since there are no other obvious exploits to take advantage of in the landscape, another approach is necessary if the previous methods are not suitable for the system designer. One approach that does not depend on the landscape and its features is Genetic algorithms. As described in section 2.4, these algorithms randomly “breed” states to create new ones instead of traversing the landscape to get new states.

A basic description of a general Genetic algorithm has been given in Figure 2.11. However, the functions that perform the basic operations of this algorithm need to be defined based on the problem, and parameters need to be set to determine the algorithm’s performance and complexity.

So, with the description of the general Genetic algorithm given in Figure 2.11, we implemented the operations defined in the algorithm description as such:

Figure 4.18: Reverse lookup of a column from a configuration X_G with $k = 2$

SEEDDED.

Input: A (defective array), N, M (array dimensions), k_{max} ,

S (time limit for Simulated Annealing), T (amount of Simulated Annealing instances),
W (width of window)

Returns: X (configuration of A)

{We begin by running *Greedy*}

$X_G \leftarrow Greedy(A, N, M, k_{max})$

{Generate the LUT}

for $j = 1 \rightarrow M$ **do**

$LUT[j] \leftarrow generateAllColumnStates(A, j, k_{max}, N, M)$

end for

{We compress X_G with the reverse look-up method}

$X \leftarrow reverseLookUp(LUT, X_G)$

{Simply call *Partial* with X as a starting point instead of the random one, if X satisfies constraints}

if $C(X) = True$ **then**

return $Partial(X)$

else

return $Partial()$

end if

Figure 4.19: Seeded algorithm

- *evaluate*: This function sorts the total population in descending order based on the

heuristic function $h(X)$ that is being used. This means that the “fittest” members of the population will be at the top of the list.

- *selection*: This function is responsible for selecting the states that will be bred to create new ones. Since we have already sorted the population, it is enough to just select a certain amount from the top of the list. This amount is exactly the top 2/3rd's (66%) of the population, and the reason for this is because of the way that the next function, *crossover*, was designed.
- *crossover*: This function is responsible for producing new states out of a selection of old ones. The way we chose to implement this is by pairing the best states according to the evaluation and producing one offspring from each pair. This means that we have a **2:1 breed ratio**. For every 2 states, 1 new is produced.

We chose to keep the population constant and use each breeder state only once, so in order to do that with a 2:1 breed ratio, the amount of breeder states had to be 2/3rd's of the initial population. The offspring produced, which is amounts to 1/3rd because of the 2:1 ratio, replaces the worst 1/3rd of the initial population. So after a crossover, we have the best 2/3rds and their offspring as the new population.

The way that the actual crossover is implemented is shown in Figure 4.20, and it is the standard [11] way of breeding two members of the population. Also shown in that figure is the mutation process, which is explained below.

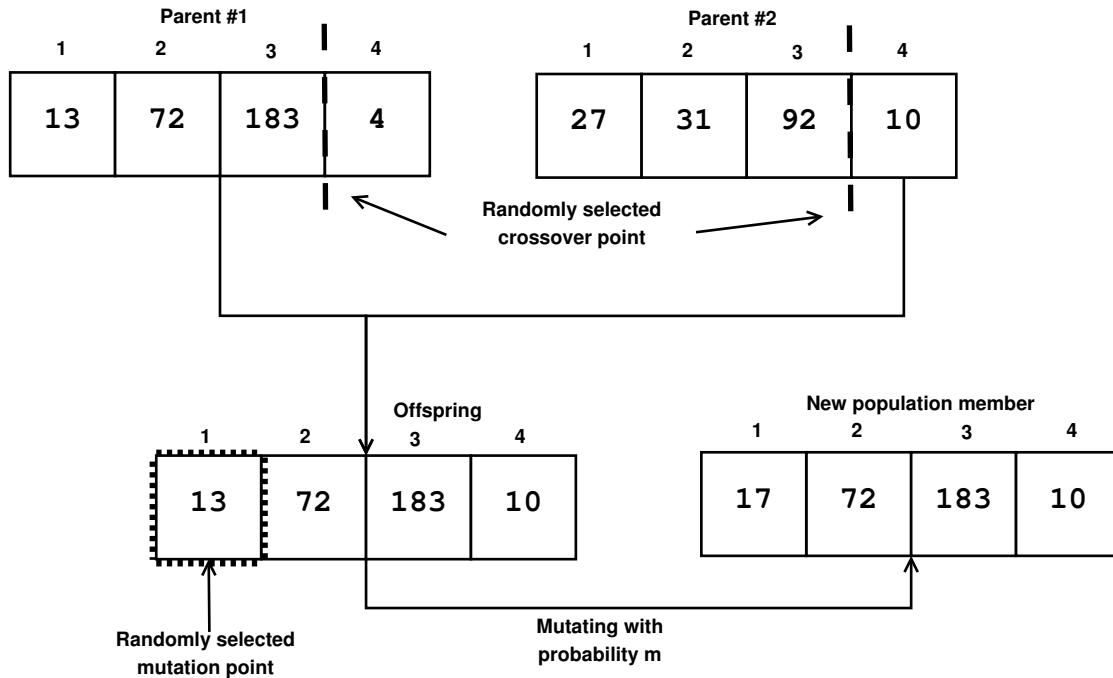


Figure 4.20: Crossover and mutation, with $M = 4$

We can see from this figure why the compressed complete state representation is very useful for use in this algorithm.

- *mutate*: This function is applied to every new offspring that is produced. With a given probability m , a random “gene” is mutated, which means it is assigned a new random (but valid) value. A “gene” in our case is the value at one index of the compressed complete state representation. The probability m basically dictates the rate at which new “genes” are injected in the selection pool, and it is generally a rather low probability. For our implementation, we used $m = 5\%$.

From the description of the general Genetic algorithm in Figure 2.11 we can see that we also need to define the amount of generations as well as the size of the initial population. These 2 are tunable parameters that define the algorithms time complexity and performance, as we will see in the evaluation in chapter 5.

4.5.8 Random search

As a last step, we implemented a completely random algorithm, mainly to use as a comparison against the other non-deterministic algorithms, to show that applying more than simple randomness has an actual benefit.

The *Random* algorithm is very simple: It performs S random steps, and it stores the best state that it encounters. After S steps have been performed, the state stored as the best is returned. S here is a tunable parameter that directly affects the time complexity of the algorithm, and potentially its accuracy, as we show in chapter 5. Figure 4.21 shows a pseudo-code description of the algorithm.

4.6 Conclusions

In this chapter we described our approach in detail, by defining the problem domain to be the combinatorial optimization domain, giving some specific details of the problem such as the expected complexity and the definition of bottleneck columns.

We formally defined three different heuristic functions with increasing amount of specific problem knowledge: Manhattan Distance, Throughput Loss and Bubble Impact. In the case of the latter, we also defined the experimental setup and described the exact process from which the approximation was extracted.

We also defined and described the heuristic search algorithms. We presented a total of 8 algorithms: *Greedy*, *BruteForce*, *FastBruteForce*, *Hybrid*, *Partial*, *Seeded*, *Genetic* and *Random*. For each algorithm we also explained the motivation behind their conception and based on this motivations also made some predictions about their performance and behavior.

```

RANDOM.
Input: A (defective array), N, M (array dimensions),  $k_{max}$ ,
S (random steps limit)
Returns: X (configuration of A)
  {Create the LUT containing all possible states per column} {Also initialize X to the
  first state per columns}
  for  $j = 1 \rightarrow M$  do
     $LUT[j] \leftarrow generateAllColumnStates(A, j, k_{max}, N, M)$ 
     $X[j] \leftarrow 1$ 
  end for
   $cbest \leftarrow X$ 
  for  $s = 1 \rightarrow S$  do
    for  $j = 1 \rightarrow M$  do
       $X[j] \leftarrow n$  {Generate a random state}
       $decompress(X, LUT)$ 
      if  $C(X) = True$  and  $best(X, cbest) = X$  then
        {Check if this new X satisfies constraints and if it is better than the currently
        stored cbest}
         $cbest \leftarrow X$ 
      end if
    end for
  end for
  return  $cbest$ 

```

Figure 4.21: Random algorithm

Evaluation

In this chapter the evaluation of the various approaches described in chapter 4 is presented. Any design choices left for the evaluation part are defined and explained. We first describe our experimental setup in section 5.1, and then we present our results along with our interpretation in section 5.2.

5.1 Experimental setup

In this section the way that the evaluation procedure was carried out is explained in order to make it reproducible. We define the inputs, the way that the benchmarking was carried out, and exactly what was measured.

5.1.1 Input sizes

The way that the problem scales with input size increase has been explained in detail in section 4.2 as a typical example of combinatorial explosion. We also mentioned in section 3.1 that we would like to keep the results as relevant as possible with industry standards. The first fact leads us to consider arrays with $N = 2^k$ processors, and the second to consider arrays of 4 and 8 processors.

The actual architecture of the processor also matters. We chose to benchmark processors with 4 and 5 pipeline stages, since the 5-stage pipeline is a rather typical reference point for a general purpose processor, and many DSP processors have 4 stage pipelines.

Also, we have already seen that the WB stage of a 5-stage pipeline processor is merely an interconnect. Taking that into account in the case of the decoupled multiprocessor array shown in Figure 3.3, we can see that this approach, as seen from this thesis' viewpoint of a processor with interconnected pipeline stages, a 5-stage processor is equivalent to a 4 stage processor with an extra interconnect stage.

For the reasons above, we chose to use the following array sizes for our benchmarking process: **4x4, 4x5, 8x4 and 8x5** multiprocessor arrays. Wherever the difference in input size (or lack thereof) is important for an evaluation result, the results will be presented comparatively in terms of the array size.

As mentioned in section 4.2, when the maximum amount of processors that can be created from an input array is equal to the initial amount of processors that were available ($k = N$), it means that there were no defects in the array. Since this thesis is about defect tolerance, that particular case is of no interest. So, the maximum value for k that we create inputs for is $N - 1$. With all of the above taken into account, we created input arrays with sizes 4x4 and 4x5 with random defects such that we could create $k = 1, 2, 3$ processors, and arrays of 8x4 and 8x5 size with random defects such that we could create $k = 1, 2, 3, 4, 5, 6, 7$ processors. This means that we have in total 20

different arrays. In the results presented below, the abbreviation $N \times M.k$ shows which array that result refers to. In the cases where only the $N \times M$ size is given, k is the free variable of the x-axis, if it applies to that result.

5.1.2 Benchmark configurations

When describing the approach in chapter 4, we defined 3 different heuristics for the heuristic search algorithms that we implemented. We also showed that some of these algorithms are parametrisable to be either faster, or more accurate. We also defined an example constraint, that can be turned on or off.

The comparison that we believe needs to be made is amongst the different algorithms, so evaluation results will be grouped by algorithms. This means that heuristics, tunings and constraint usage define different benchmark configurations. Since we have 3 different heuristics, 2 different tuning settings (fast or accurate) and 2 options for constraints (on or off), we define 12 different benchmark configurations, as shown in Figure 5.1. Each configuration is given a abbreviation, in order to make the results more readable.

Tuning	Constraints	Heuristic	Abbreviation
Accurate	Off	Manhattan Distance	Acc-Off-MD
		Throughput Loss	Acc-Off-TL
		Bubble Impact	Acc-Off-BI
	On	Manhattan Distance	Acc-On-MD
		Throughput Loss	Acc-On-TL
		Bubble Impact	Acc-On-BI
Fast	Off	Manhattan Distance	Fast-Off-MD
		Throughput Loss	Fast-Off-TL
		Bubble Impact	Fast-Off-BI
	On	Manhattan Distance	Fast-On-MD
		Throughput Loss	Fast-On-TL
		Bubble Impact	Fast-On-BI

Figure 5.1: The 12 different benchmark configurations

While the On/Off setting of constraints is straightforward, and the usage of a specific heuristic as well, the actual parameters of a tuning need to be set. The impact of each parameter in each algorithm is explained at the algorithm definitions in section 4.5. The exact values that we used for each tuning are shown in Figure 5.2.

5.1.3 Measured quantities

After having defined what the input is and how the benchmarks are configured, the quantities that will be measured need to be defined as well. The definition and reason to measure each quantity are given here. The actual measurements are presented in the next section.

1. **State space size:** It has been mentioned a few times already that state space size is a major obstacle that we have to work against. The formulas for calculating it

<i>Accurate</i>		<i>Fast</i>	
Algorithm	Parameters	Algorithm	Parameters
<i>Greedy</i>	N/A	<i>Greedy</i>	N/A
<i>BruteForce</i>	N/A	<i>BruteForce</i>	N/A
<i>FastBruteForce</i>	N/A	<i>FastBruteForce</i>	N/A
<i>Hybrid</i>	$T = 1000$ $S = 300$	<i>Hybrid</i>	$T = 100$ $S = 20$
<i>Partial</i>	$T = 1000$ $S = 300$ $W = 10^6$	<i>Partial</i>	$T = 100$ $S = 20$ $W = 10^6$
<i>Seeded</i>	$T = 1000$ $S = 300$ $W = 10^6$	<i>Seeded</i>	$T = 100$ $S = 20$ $W = 10^6$
<i>Genetic</i>	$G = 1000$ $P = 60$	<i>Genetic</i>	$G = 100$ $P = 30$
<i>Random</i>	$S = 10^4$	<i>Random</i>	$S = 10^3$

(a) “Accurate” tuning parameters

(b) “Fast” tuning parameters

Figure 5.2: The algorithm parameters for each tuning

have been given in section 4.2, but we also measure the actual numbers that our inputs produce, to get a more realistic picture of what this combinatorial explosion means.

2. **Memory use:** When describing the look-up table technique that enabled the easier implementation of some of the algorithms, it was mentioned that the cost to pay was extra memory. To see just how much this extra cost is, we measure the memory consumption of the algorithms.
3. **Execution time:** One of the most important aspects of the algorithms to measure is the actual time needed to provide a solution to the inputs given. The time that it took for each algorithm to solve a particular input from start to finish was measured¹, under all the different benchmark configurations for all inputs.
4. **Solution quality & accuracy:** The other part of the trade-off after execution time is the quality of the solution and its accuracy. The distinction here is that by quality we mean just how good the solution is in terms of absolute heuristic evaluation, and by accuracy we mean how close it was to the actual global optimal solution, wherever we could apply exhaustive search.
5. **Simulated multiprocessor array performance:** Since we had the setup described in section 4.3 to create processors with any bubble configuration, and to

¹Execution time was evaluated on an Intel[®] Core[™]2 T7200 CPU clocked at 2GHz, using a single core (i.e. no inherent parallelism in the algorithms has been exploited).

simulate program execution on these processors, we decided to also simulate processors that would result from applying the algorithms to some of the input arrays and compare them to an ideal reconfiguration, to see just how well they would perform in as close to actual implementation conditions as possible.

It is also worth noting that in order to maintain statistic validity, measurements of quantities that do not maintain the exact same value at every execution (such as execution time and solution quality & accuracy), were averaged over 10 repeated measurements.

5.2 Evaluation results

In this section we present the resulting figures from the evaluation process. They are presented in the categorization that was described in the previous section.

5.2.1 State space size

The first and one of the most important factors to measure is the exact size of the state space. In order to do that the only thing that is needed is the actual input, and the formulas presented in section 4.2. We measured the state space size for all input array sizes (4x4, 4x5, 8x4, 8x5) with the free variable being the amount of working processors that could be created based on the input array. The results are presented in Figure 5.3.

It is worth noting the logarithmic scale used on the y-axis. We can clearly see the effect of the combinatorial explosion in this plot: Within 7 steps the problem grows from about 10 states to 10^{19} . These numbers should be kept in mind when discussing the accuracy of algorithms, given that they have to operate within such vast state spaces.

5.2.2 Memory use

We also discussed that in order to facilitate some of the search algorithms, a look-up table technique had to be implemented. This look-up table provides a number of features that are essential for the search algorithms, as described in section 4.5, but it comes at the cost of extra memory usage. In Figure 5.4, we show exactly how much extra memory is needed for this look-up table, compared against the memory usage of algorithms that do not make use of a look-up table (marked in the plot as “regular”). The results are presented for each array size.

As with the state space size comparison, the free variable here is the amount of working processors that could be created on the respective input array. The scale is logarithmic again, and the units are bytes. We see that memory usage has an exponential growth rate, but it remains within easily manageable sizes for today’s standards (about 2 MB for the largest input).

We also see that “regular” (non look-up table) algorithms use a near constant amount of memory. In the case that memory is indeed a concern, an algorithm of that type could be considered, so the look-up table technique although useful, is not the only way to go.

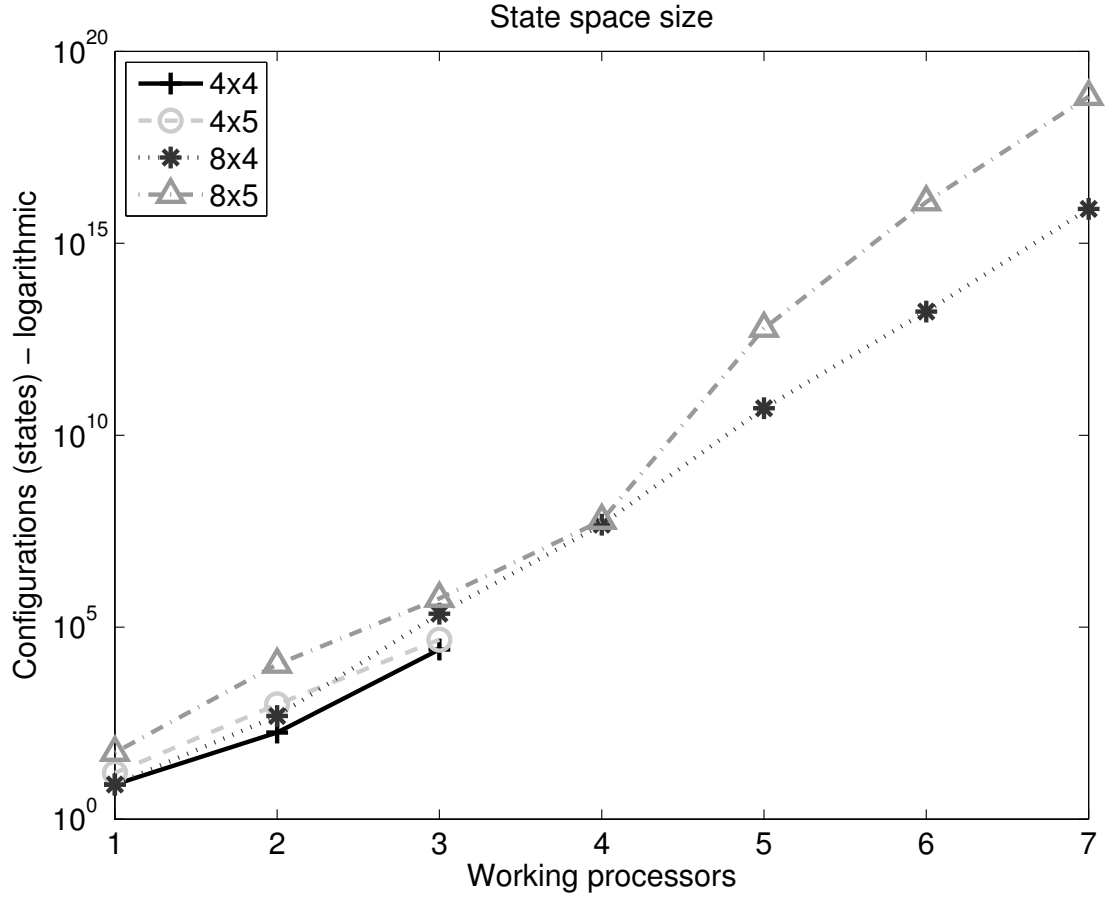


Figure 5.3: State space size

5.2.3 Execution time

Execution time, as already discussed, is one end of a fundamental trade off in the design of the algorithms we implemented. We can tune each algorithm to be fast, but we sacrifice solution quality & accuracy. So the results for these trade-off quantities need to be considered within this context.

This is one of the reasons that different benchmark configurations were created. Since there are 12 different benchmark configurations, and 4 different array sizes, normally we would have to include 48 execution time comparisons. However, in order to make the results more readable, we will show those results that carry important information. We will use the 8x5 array size as a case study, simply because the focal point of the comparison is among algorithms and not among array sizes. An extensive presentation of all results containing all time comparisons and all array sizes can be found in appendix A.

Before proceeding to an actual comparison between the algorithms, we need to present the impact of each option in the benchmark configuration, to establish proper context when comparing algorithms under a certain configuration.

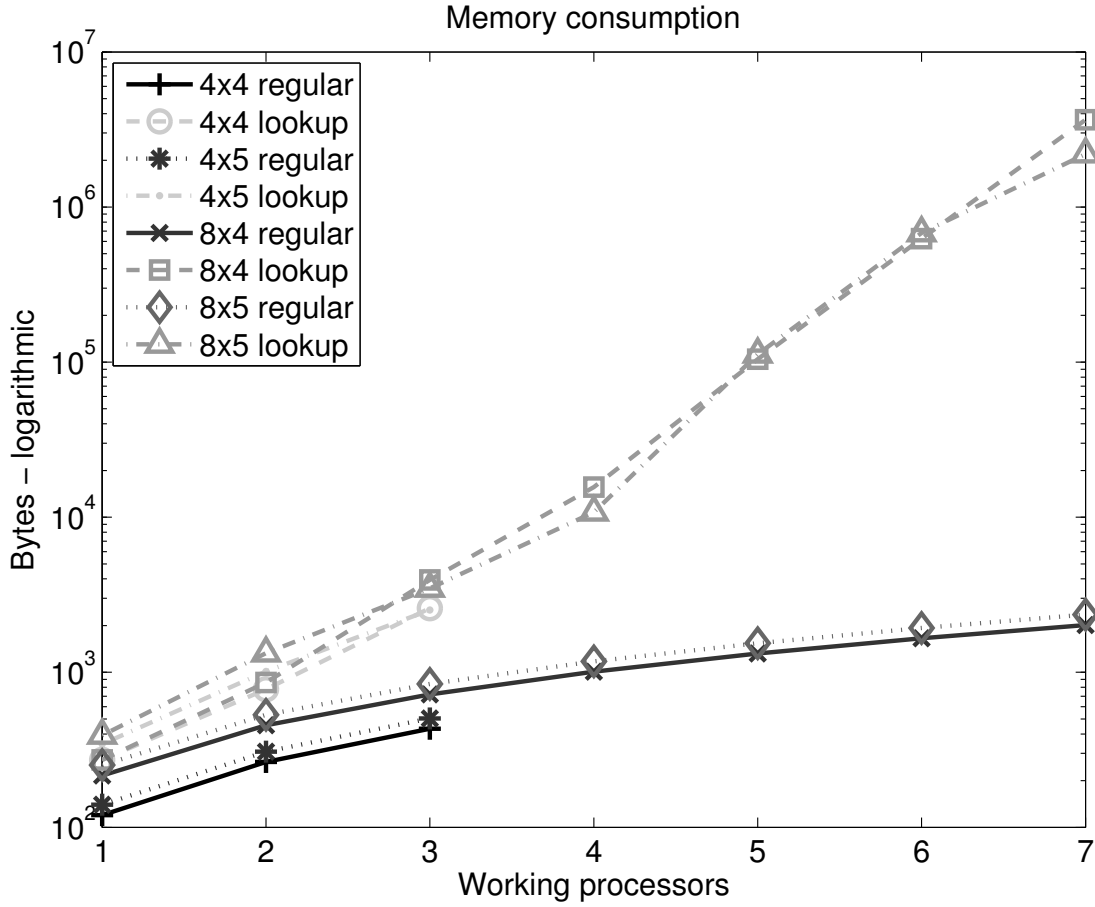


Figure 5.4: Memory consumption

5.2.3.1 Tuning impact

For the 8x5 size there are 12 different execution time comparisons to show. As discussed, there are 2 main options that heavily affect execution time: Either go for a “fast” tuning, or for an “accurate” one. We can witness this by keeping the rest of the options (constraint usage and heuristic function) in the benchmark configuration the same, and comparing the results of the fast and the accurate tuning. For the sake of this comparison, we will use the Manhattan Distance heuristic, without any external constraints applied. Figure 5.5 shows a side-by-side comparison of the Acc-Off-MD versus the Fast-Off-MD results.

The first thing to note about both plots is the usage of a logarithmic y-axis. This is mostly to accommodate for the exponential time complexity of the exhaustive search algorithms. It can be seen in both plots that they have a very steep exponential growth. It is actually exactly the same, since these algorithms do not have any tunable parameters. Also the same in both plots is the execution time for the *Greedy* algorithm, since it too does not have any tunable parameters. We can also see that as expected, *Greedy* is **much faster** than all other algorithms. These conclusions apply to all configurations:

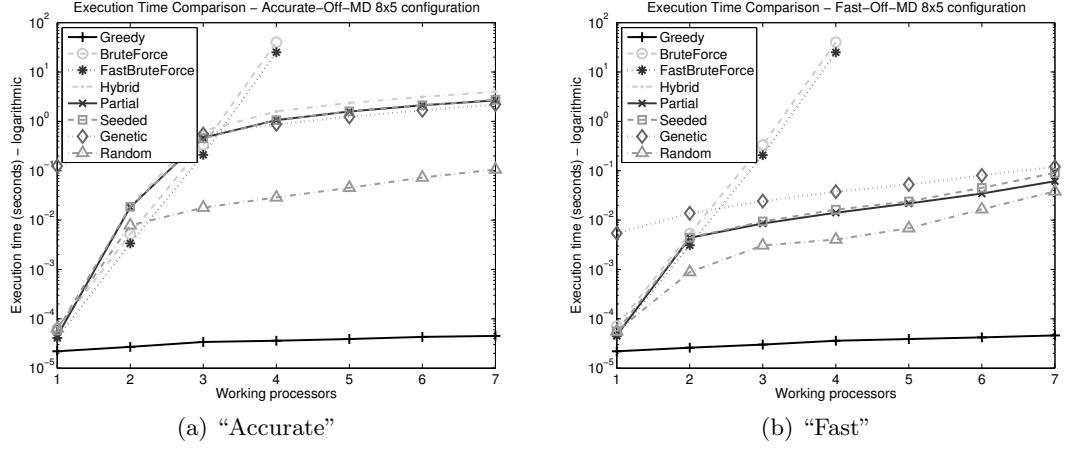


Figure 5.5: Acc-Off-MD vs Fast-Off-MD (execution time)

The exhaustive search algorithms are always the slowest, while *Greedy* is consistently the fastest algorithm.

As for the comparison between the actual tunings, the conclusion is pretty obvious: The algorithms simply become slower when tuned for more accuracy, exactly as expected.

5.2.3.2 External constraint impact

We can now investigate the impact of the second option in benchmark configuration, which is the usage of external constraints. The option here is whether to impose them or not. So, keeping the rest of the options the same, we will show a comparison between two configurations. We will use the "fast" tuning set and the Throughput Loss heuristic function for this purpose. Figure 5.6 shows a side-by-side comparison of the Fast-Off-TL versus the Fast-On-TL benchmark configurations.

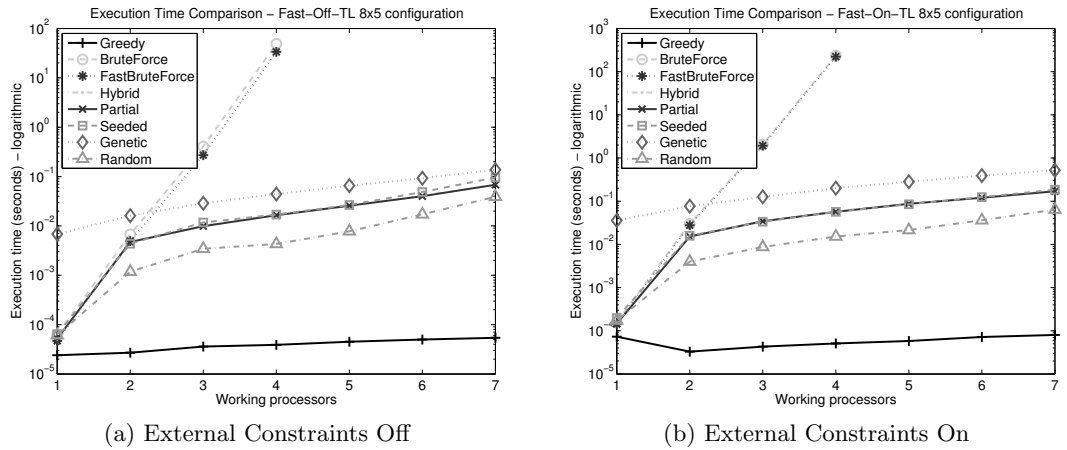


Figure 5.6: Fast-Off-TL vs Fast-On-TL (execution time)

While it may not be obvious at first glance, there is an impact: The execution time for all algorithms has been increased by **about one order of magnitude**. All algorithms have been affected in pretty much the same way, except for the *Greedy* algorithm, which is expected, because as discussed at the definition of this algorithm, it cannot take external constraints into account except for when it has actually completed its execution. We can also see here that the conclusions from the previous comparison also apply: *Greedy* is still the fastest algorithm in both cases, and the exhaustive search algorithms are always the slowest. External constraint usage simply increases the execution time of all algorithms by about one order of magnitude. This comes from the fact that examining whether a state violates a constraint or not is an extra calculation that needs to be done at each state that the algorithms produce.

5.2.3.3 Heuristic function impact

The last option in the benchmark configuration is the choice of the heuristic function. Here the available choices are as many as the heuristic functions we defined, which are three: Manhattan Distance ($h_{MD}(X)$), Throughput Loss ($h_{TL}(X)$) and Bubble Impact ($h_{BI}(X)$). In order to see how the choice of each function affects execution time, we will keep the other two options constant and compare the execution time of algorithms with three different heuristic functions. For this purpose, we will use the “accurate” tuning set and the “off” option for constraint usage. Figure 5.7 shows the 3-way comparison amongst the different benchmark configurations: Acc-Off-MD, Acc-Off-TL, and Acc-Off-BI.

We can see from this 3-way comparison that practically the choice of heuristic function **does not affect** execution time. It is mostly a choice that will affect, as we will show next, the quality and accuracy of the algorithms. We can see also that the conclusions extracted before, still apply: *Greedy* is the fastest choice, while the exhaustive search algorithms are the slowest.

5.2.3.4 Algorithm comparison

Now that we have shown the impact of each choice in the benchmark configuration on execution time, we can compare algorithms to one another, presenting the comparison for just one benchmark configuration. The results of all benchmark configurations as well as array sizes can be found in appendix A

Since we know that: a) the tuning set affects all tunable algorithms in the same way, b) the same applies to the usage of constraints, and also that c) the choice of heuristic function has no impact on execution time, we can practically choose any benchmark configuration to compare the execution time amongst algorithms. For this reason we will present the execution time results of the Acc-Off-BI benchmark configuration, and the conclusions will apply for all 12 benchmark configurations. We have already reached the conclusion that the exhaustive searches are the slowest ones. In order to see the comparison of the rest of the algorithms better, we will omit these algorithms from the following comparison in order to be able to use a linear time scale and see exactly each algorithm’s results. Figure 5.8 shows this comparison of algorithms in linear time.

We can now make the following observations:

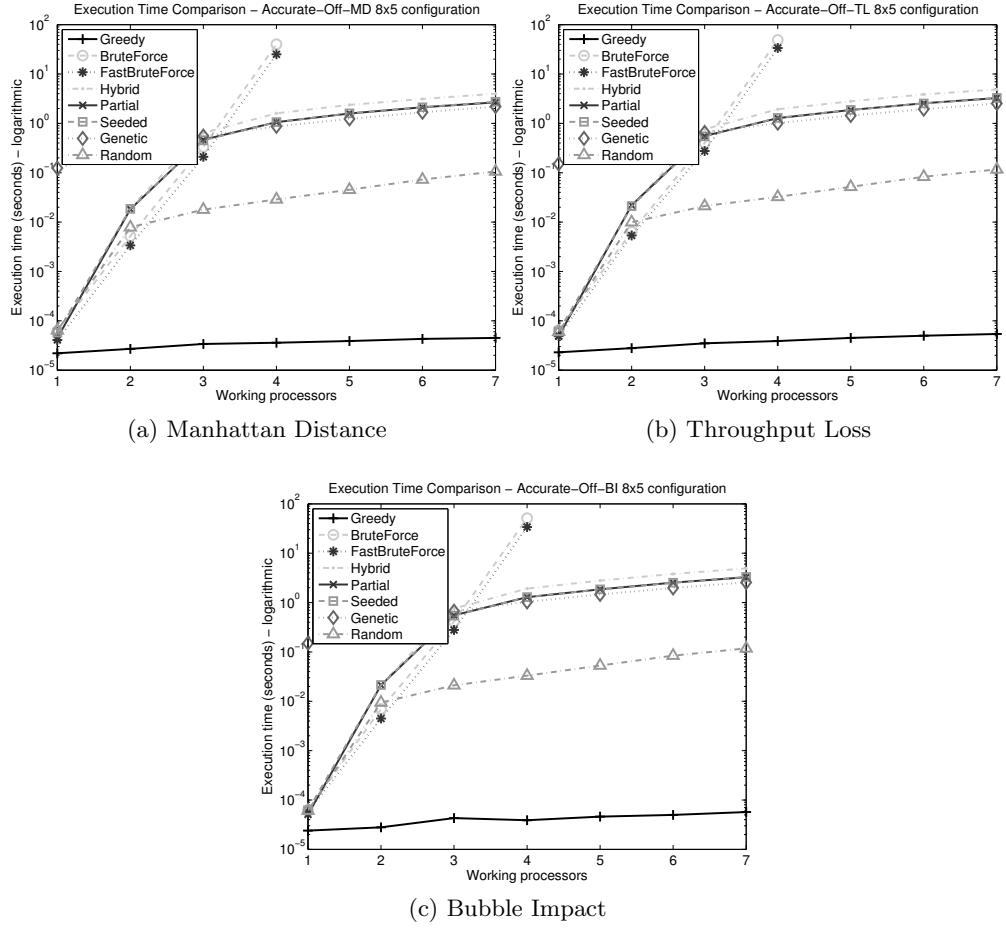


Figure 5.7: Acc-Off-MD vs Acc-Off-TL vs Acc-Off-BI (execution time)

- *Random* is faster than all the others except *Greedy* and has almost a constant speed. This is expected since the time complexity of *Random* is dependant on the steps that it is set to make, not the input size (working processors).
- *Hybrid* performs slower than all the others. This is expected because of the fact that, even though it tries to work around the size of the state space by using a blend of Simulated Annealing and exhaustive search, it still scales based on that size.
- On the other hand, both *Partial* and *Seeded* perform somewhat faster. This is because both of these algorithms scale on their pre-defined window size, which is just a fraction of the state space size. The scaling that we see here comes actually from the inherent scaling in complexity that the creation of the look-up table induces. This is why the steepness of these algorithms is smaller than that of the *Hybrid* algorithm.
- The execution times of *Partial* and *Seeded* directly trace each other, almost exactly

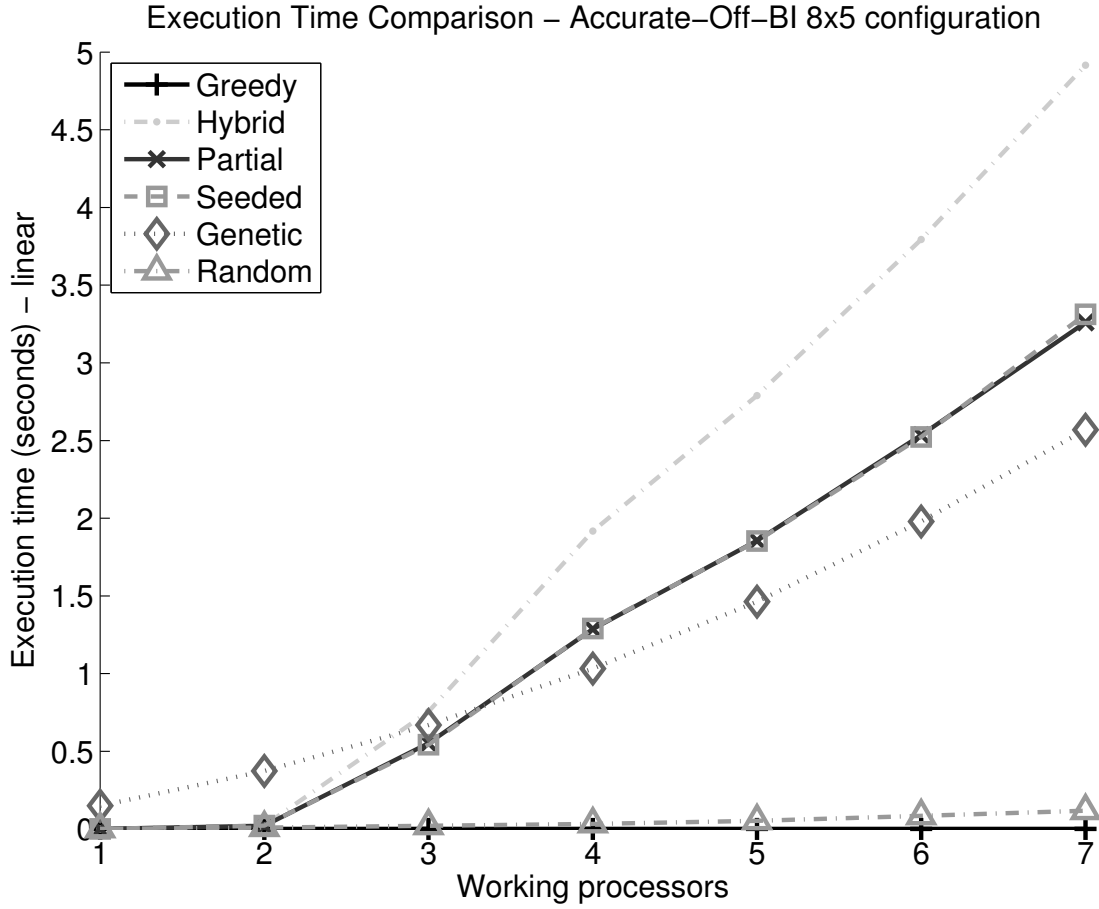


Figure 5.8: Execution time comparison, Acc-Off-BI configuration

coinciding. This comes as no surprise, since as was described in the definition of *Seeded*, in terms of implementation it is actually just the *Partial* algorithm preceded by a call to *Greedy* to get the seeding point and a reverse look-up. Since *Greedy* has such a low execution time, it does not affect the total execution of *Seeded* by much in comparison to *Partial*.

- Finally, we can make the following observations about *Genetic*: It seems that for smaller problem sizes (1 and 2) it is actually the slowest one. In fact, it is even slower than the exhaustive search algorithms, as seen in previous comparisons. This is again expected, because the actual scaling variable of the *Genetic* algorithm is not the input size, but rather its parameters (generations G and population size P). This is why we see it being so slow for the first few input sizes. However, this quickly turns into an advantage, as it becomes the fastest algorithm after *Greedy* and *Random*. The scaling seen in this figure comes again from the inherent scaling by input size of the creation of the look-up table.

5.2.4 Solution quality & accuracy

While execution time as a concept is pretty straightforward, solution quality and accuracy need defining. By **solution quality** we mean the **actual heuristic value** of the solution that an algorithm provided. This means that the numbers between different heuristics are not comparable. By **solution accuracy** we mean the **percentage differential** between the heuristic value of the solution provided by an algorithm against the one that exhaustive search provided. This means that *by definition*, exhaustive search algorithms have 100% accuracy. So, this comparison is only useful for the algorithms other than the exhaustive search ones.

Since we cannot directly compare different heuristic values from different functions, the options in the benchmark configurations that we can see the impact of are the tuning sets and the usage of constraints. The different heuristic functions will have to be separately compared in the final algorithm comparison at the end. However, since accuracy by definition hides the specifics of a heuristic, we can use that as a comparison when determining the impact of heuristic function choice in algorithm accuracy.

5.2.4.1 Tuning impact

The two available choices here are the “fast” and “accurate” tuning settings. They were designed to have a direct and uniform impact on all algorithms towards the respective end of the trade-off. We saw previously that accuracy had a direct impact on speed, and now we want to see if the same holds true the other way round. For this reason, we will keep the other two options of the benchmark configuration constant, and compare between tuning settings only. We will use the “off” setting for constraint usage, and the Manhattan Distance heuristic for this. Figure 5.9 shows a side by side comparison of the Acc-Off-MD versus the Fast-Off-MD benchmark configurations for solution quality.

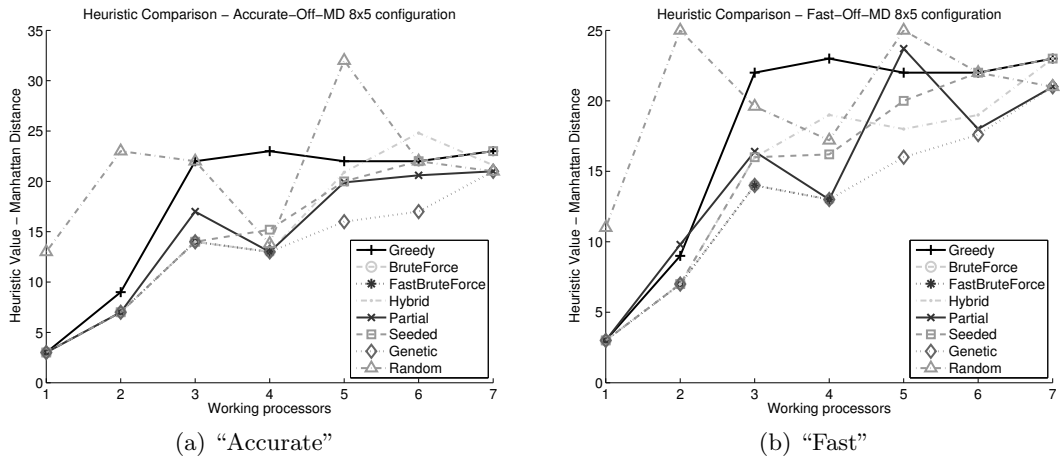


Figure 5.9: Acc-Off-MD vs Fast-Off-MD (heuristic value)

Interpreting the impact here might not be such a simple task, because of the complexity of the picture shown, which comes from the fact that the precise value depends

not only on the size, but also on the actual defect distribution over the array. Things become more clear when we view the same comparison but in terms of **accuracy**. This is as defined above, the **percentage difference** of the heuristic value of an algorithm against the exhaustive search value. This implies that we are only able to do this comparison for the input sizes that we could apply exhaustive search to, namely $k \leq 4$, as discussed in section 5.1.

The following comparison, Figure 5.10 shows the same comparison as above but expressed in terms of accuracy. This makes things easier for interpreting the results. Tuning has a **direct** impact on accuracy, as designed. In the “Accurate” tuning set, algorithms maintain a higher accuracy, consistent as the input size increases, while on the “Fast” tuning set, algorithms quickly start to deteriorate in accuracy. This is exactly the effect that the tuning sets were designed to have on the algorithms. One exception seems to be *Genetic*, which does not seem to lose any accuracy regardless of tuning, and also maintains a consistent 100% accuracy for both sets, a behaviour that can be attributed to the nature of *Genetic*, which ignores inherent difficulties in a landscape by definition.

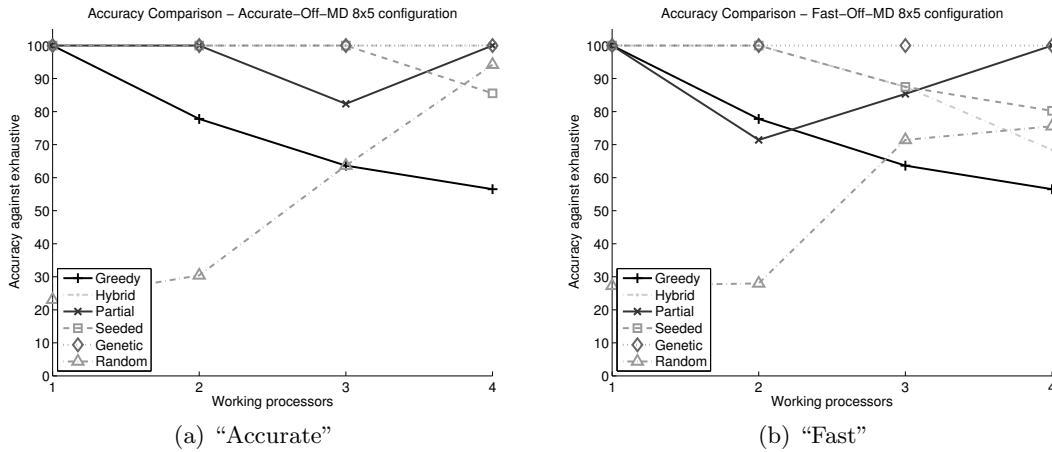


Figure 5.10: Acc-Off-MD vs Fast-Off-MD (accuracy)

5.2.4.2 External constraint impact

The other option that we can explore to see the impact of on the quality and accuracy of algorithms is the usage of constraints. By definition, this option is expected to have an impact on both quantities, since whenever an algorithm cannot provide a solution that satisfies constraints it is considered a failure. For that purpose, we keep the other two configuration options constant and compare between the two values for constraint usage, on and off. We use the “Accurate” tuning set and the Throughput Loss heuristic. Figure 5.11 shows a side by side comparison of the Acc-Off-TL versus the Acc-On-TL benchmark configuration for solution quality.

This time it is a bit easier to see the effect of external constraints on the quality of the solutions provided: When the constraints are on, most algorithms start to produce

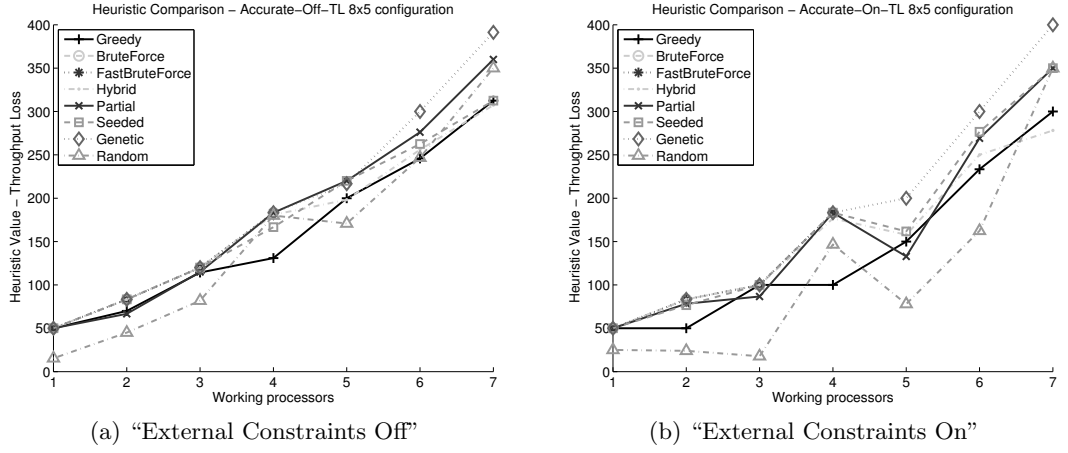


Figure 5.11: Acc-Off-TL vs Acc-On-TL (heuristic value)

solutions with lower heuristic values. In the case of this heuristic function, Throughput Loss, where we defined it to be a maximization problem, lower heuristic value means worse potential accuracy.

We use the term “potential” because it is actually possible that the constraints are so strict that make it impossible even for exhaustive search to find a satisfying solution. Figure 5.12 shows the previous comparison expressed as accuracy against exhaustive. As expected, the presence of constraints affects the accuracy of the algorithms. Interestingly, this impact is not big: Algorithms that could find a solution without constraints seem to find a solution within constraints with almost the same accuracy as before, with one notable exception: *Greedy* seems to suffer the most from constraints. This is of course expected since by its definition, *Greedy* cannot predict whether the solution it is constructing iteratively will violate a constraint or not before it has completed its execution.

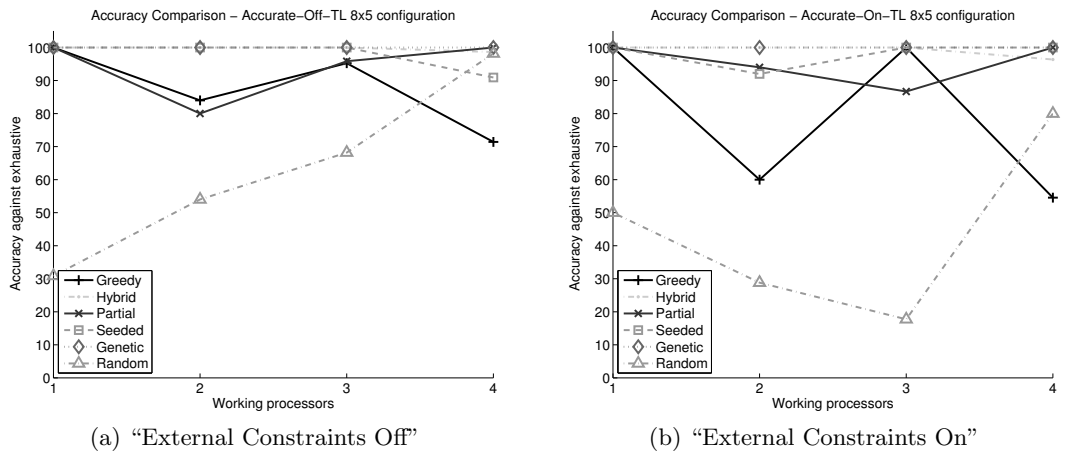


Figure 5.12: Acc-Off-TL vs Acc-On-TL (accuracy)

5.2.4.3 Heuristic function impact

The definition of **solution accuracy** hides the “units” in which heuristic functions express their values, so we can directly compare the accuracy of the algorithms when using different heuristic values. This is especially interesting because different heuristic values will form different landscapes, which is a determining factor in algorithm accuracy.

So, in order to compare the impact of heuristic functions on algorithm accuracy, we must keep the other two configuration options constant. We use the “Fast” tuning set and the “Off” setting for constraint usage. Figure 5.13 shows the 3-way comparison between the benchmark configurations Fast-Off-MD, Fast-Off-TL and Fast-Off-BI for solution accuracy.

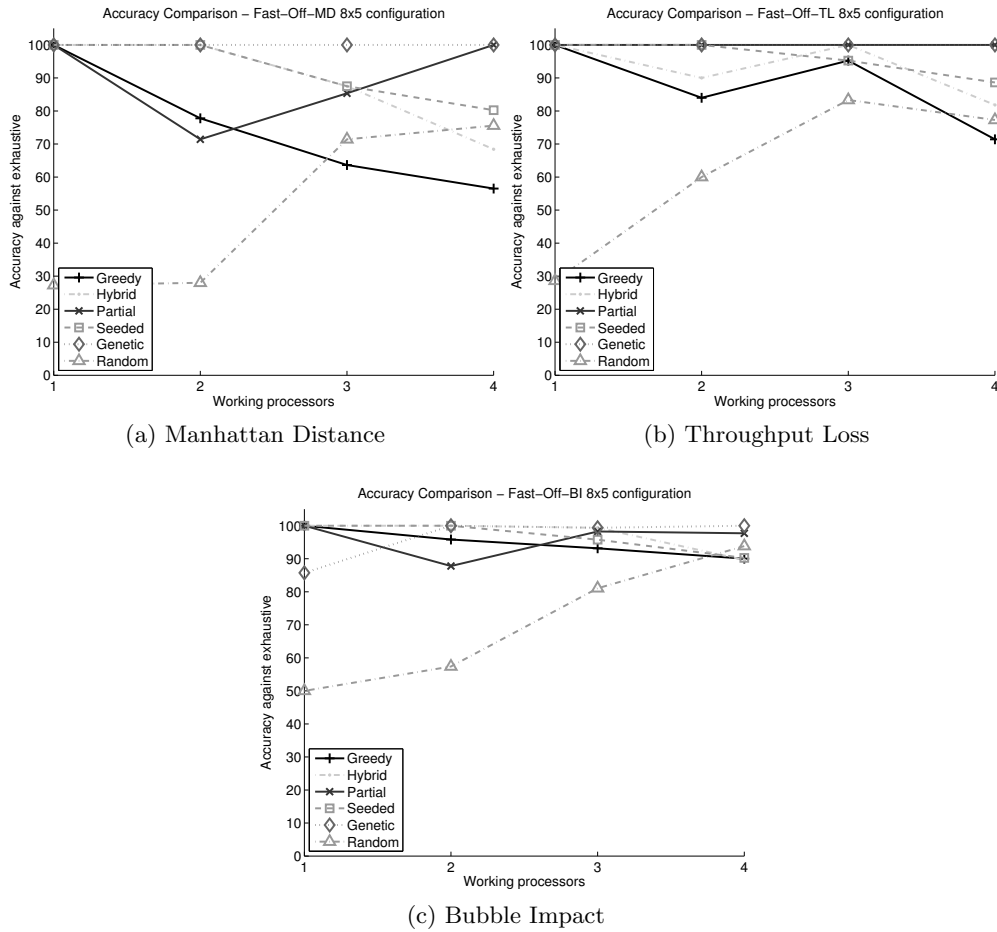


Figure 5.13: Fast-Off-MD vs Fast-Off-TL vs Fast-Off-BI (accuracy)

As expected, different heuristics make the algorithms behave differently with regard to accuracy, and the impact is different for each algorithm. So, in order to compare the algorithms, we have to treat each heuristic function as a different comparison.

5.2.4.4 Algorithm comparison

Since, as shown previously, the impact of tuning is uniform across different algorithms, and the same can be said for constraints, these two choices for benchmark configuration do not really matter when discussing solution quality & accuracy, in the sense that conclusions made for either option can be applied to the others.

However, as we also showed, the choice of heuristic function affects each algorithm individually in a different way. So, in order to keep comparisons valid and conclusions consistent, we need to examine each heuristic function on its own with regard to quality & accuracy.

To do so, we will use the same options for tuning and comparison usage in all three heuristic function comparison sets, namely the “Fast” tuning and the “Off” option for constraint usage.

Manhattan Distance

The following comparison set shows the benchmark configuration Fast-Off-MD, this time with the focus being the actual performance of the algorithms compared to each other, both in terms of solution quality and solution accuracy.

Figure 5.14 shows the solution quality of each algorithm (the Manhattan distance heuristic value) as a function of input size. It is important to remember that in this case we are talking about a **minimization** problem, so lower value is better.

We already explained that in terms of absolute heuristic value, it is hard to judge algorithm performance based on just input size. The defect distribution plays a major role in this. However, we can make some preliminary observations from this plot before we proceed to the actual accuracy comparison:

- *Genetic* is **strictly** better than every non-exhaustive algorithm.
- *Random* behaves rather erratically in terms of solution quality, which is expected given the algorithm’s definition.
- A very interesting result is the also erratic behavior of *Partial* in combination with the fact that it differentiates quite a bit from *Seeded*. This can be attributed to the fact that *Partial* has a completely random starting point, whereas *Seeded* has the consistent result of *Greedy* as a starting point.
- We can also see that *Seeded* behaves quite consistently, which means that the impact of the seeding point matters quite a bit for both *Partial* and *Seeded*.
- It seems also that *Seeded* for the most part improves its starting point, which is the solution of *Greedy*.
- *Greedy* seems to generally provide a worse solution than other algorithms.

These observations should become more definite when the comparison is presented in terms of accuracy against the exhaustive algorithms, presented in Figure 5.15. Here we can make some more specific observations:

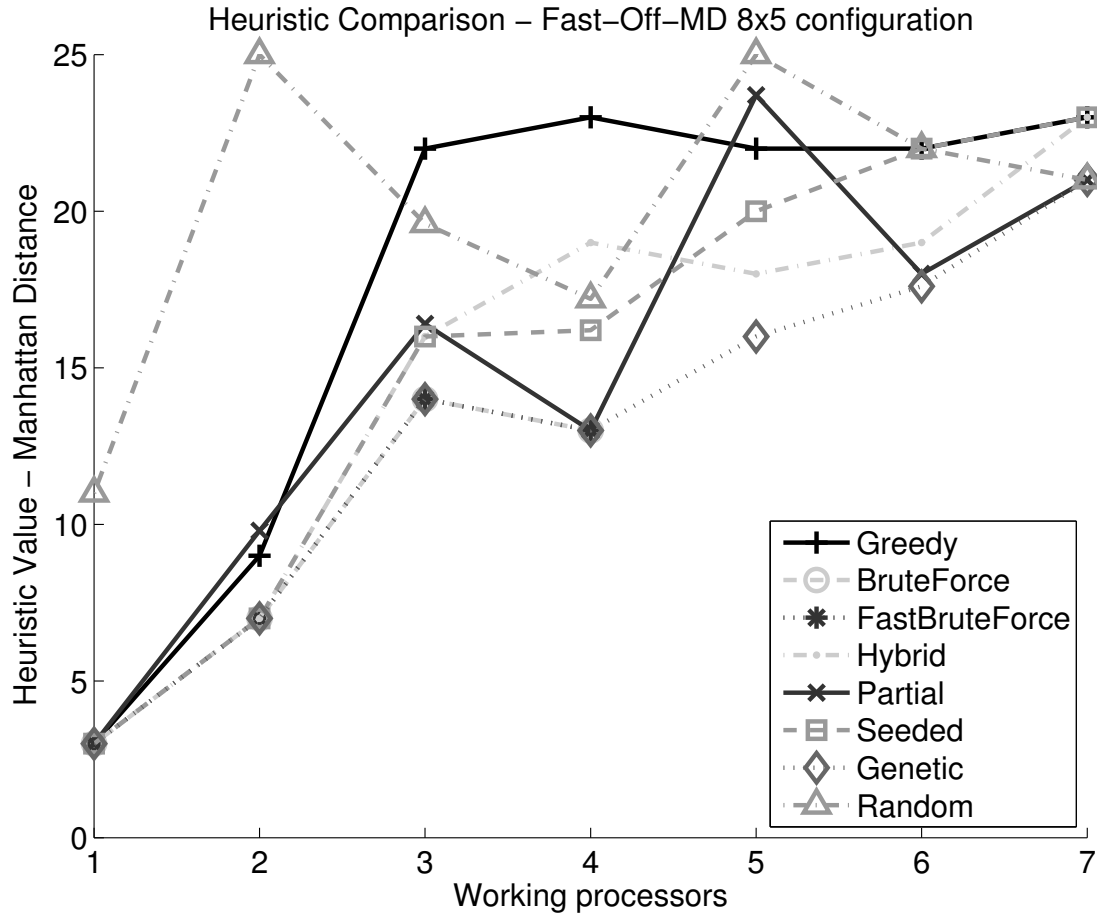


Figure 5.14: Solution quality (heuristic value) comparison, Fast-Off-MD configuration

- *Genetic* consistently proves to be 100% accurate.
- *Random* has a rather low and inconsistent accuracy.
- *Partial* is indeed inconsistent in accuracy, but still maintains a rather high amount of relative accuracy.
- *Greedy* shows a consistent decline in accuracy as problem size increases, and proves to be the least accurate algorithm compared to the other algorithms, barring the inconsistent behavior of *Random* which could make it worse than *Greedy*.
- *Seeded* always improves upon the accuracy of *Greedy*, **which proves that the idea for implementing *Seeded* indeed holds ground**; a better solution is usually a few steps around *Greedy*.
- *Hybrid* seems to be generally just as accurate as *Seeded*, but not consistently, showing that focusing on a smaller window of states with a good starting point is indeed a beneficial strategy.

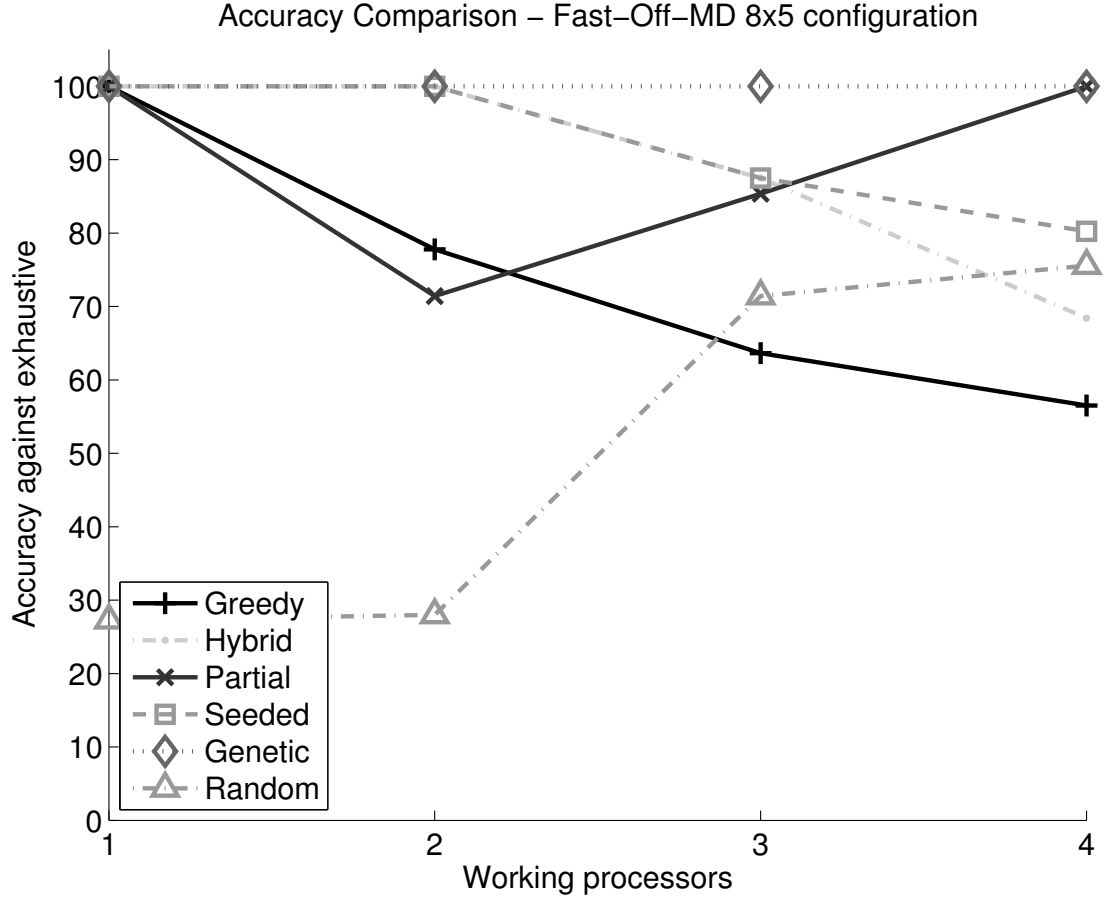


Figure 5.15: Solution accuracy comparison, Fast-Off-MD configuration

Throughput Loss

We now have to repeat the same comparison for the Throughput Loss heuristic function, both for solution quality but also accuracy. Throughput Loss defines the problem as a maximization problem, so for the following comparison in Figure 5.16, a larger value indicates a solution of better quality.

For this heuristic function it would seem that the landscape it creates affects all algorithms in about the same way. However, some preliminary observations can be made:

- *Genetic* seems to be strictly better than all other non-exhaustive algorithms.
- *Random* does not show an erratic behavior, indicating an increased amount of close to optimal local maxima in the state space.
- *Seeded* seems to have a better quality than *Greedy*.
- *Hybrid* seems to be close to optimality as well.

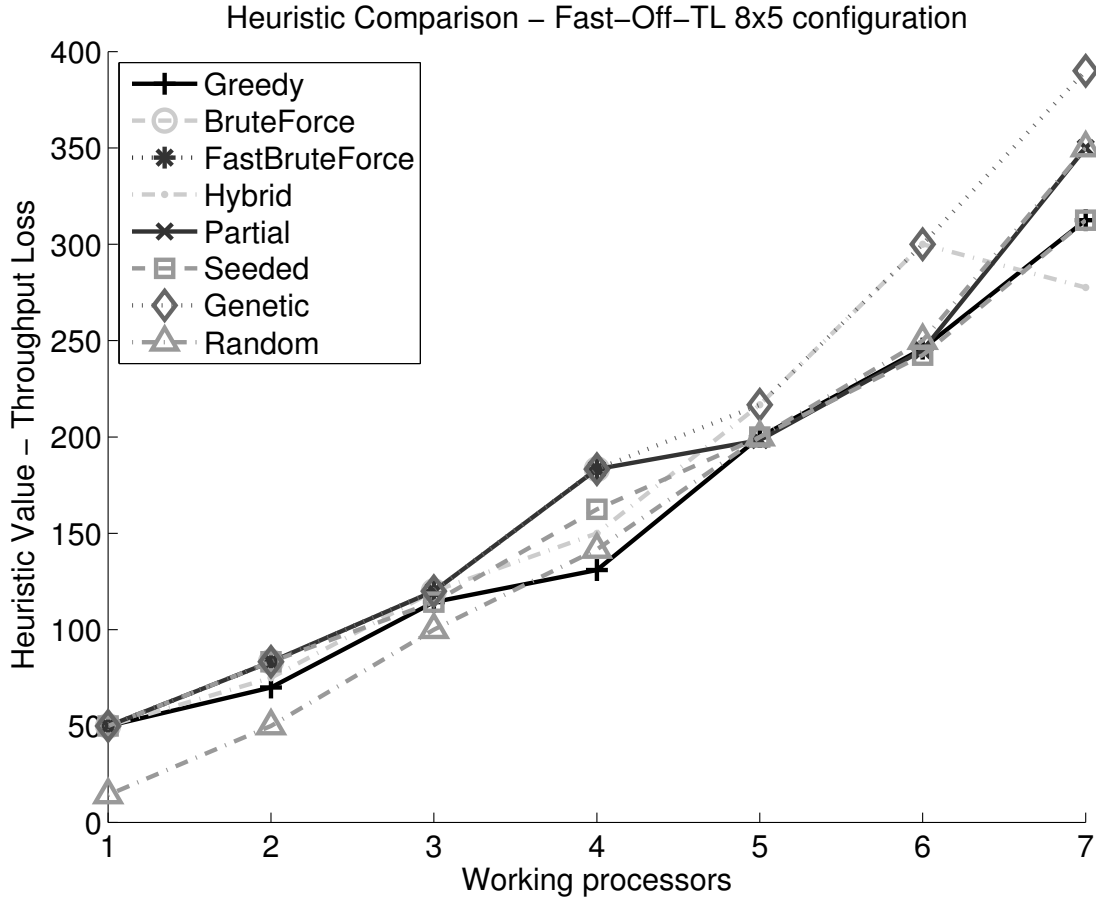


Figure 5.16: Solution quality (heuristic value) comparison, Fast-Off-TL configuration

In order to view the situation in more detail and consistency, we also need to see the same comparison in terms of accuracy, as shown in Figure 5.17.

Here some more specific observations can be made:

- *Genetic* and *Partial* show 100% accuracy.
- *Hybrid* also shows close to 100% accuracy.
- *Seeded* shows consistent improvement upon *Greedy*, and also very close to 100% accuracy.
- *Random* shows low and inconsistent accuracy as expected.
- With the exception of the expected low accuracy of *Random*, *Greedy* seems to be the least accurate algorithm, with a decaying trend.

Bubble Impact

The last heuristic function that we can compare algorithms with is Bubble Impact. As

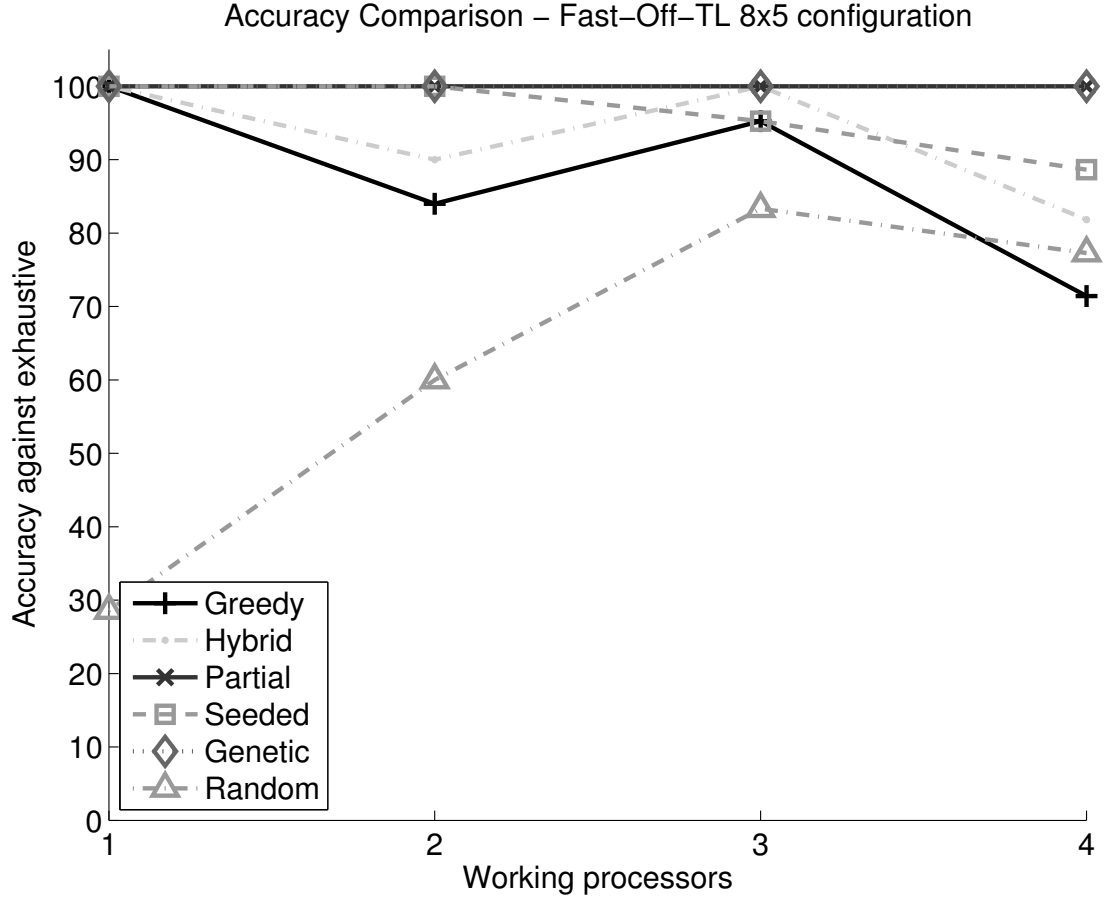


Figure 5.17: Solution accuracy comparison, Fast-Off-TL configuration

with Throughput Loss, this function describes the problem as a maximization problem, which means that a larger heuristic value implies a higher solution quality. Figure 5.18 shows a comparison amongst algorithms in terms of solution quality.

As with Throughput Loss, this heuristic function's landscape seems to affect all algorithms in the same way. Some preliminary observations can be made:

- *Random* seems to be consistently worse than other algorithms.
- *Greedy* interestingly seems to provide the best results.
- It seems that there is not much differentiation in the quality of the solution amongst the rest of the algorithms.

For a more detailed investigation, we need to study the comparison expressed in terms of accuracy, as shown in Figure 5.19. Some more specific observations that can be made are:

- Indeed, all algorithms seem to perform about as well, with the exception of *Random* which, however, appears to be improving with larger input size, a fact that can be

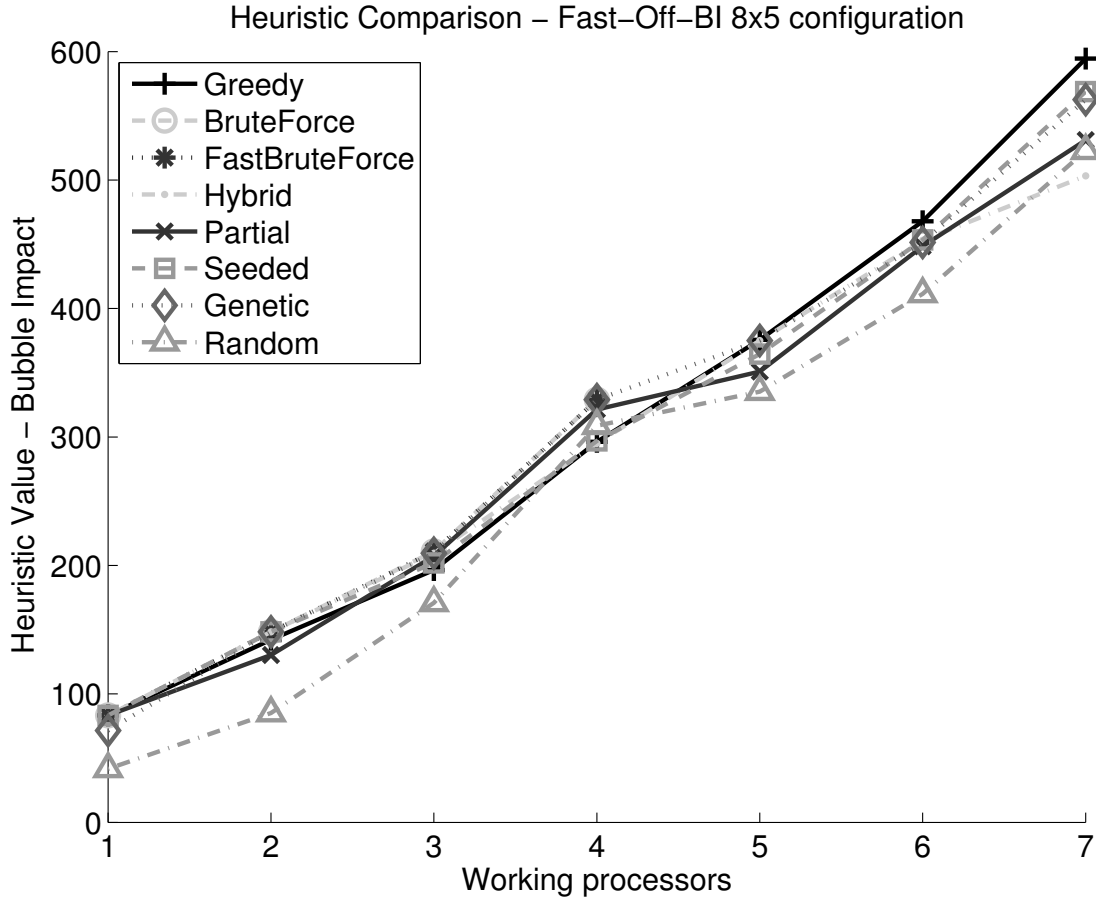


Figure 5.18: Solution quality (heuristic value) comparison, Fast-Off-BI configuration

attributed to a more smaller “period” of the landscape created by this heuristic (i.e., the chunk that seems to be repeating itself is smaller and appears more frequently than in other landscapes created by the other two heuristics).

- *Greedy*, *Genetic*, *Hybrid* and *Seeded* seem to be the best ones.
- *Seeded* proves to be better than *Partial*.

Cross-heuristic observations about the algorithms

Now that we have observed and studied the behavior of the algorithms under different configurations and different landscapes, we can make some remarks about each algorithm in terms of quality & accuracy:

- *Greedy* has turned out to be, in general, the least accurate solution, in different landscapes and especially in the presence of constraints. However, it still maintains in absolute terms a rather good accuracy, at least above 70%.

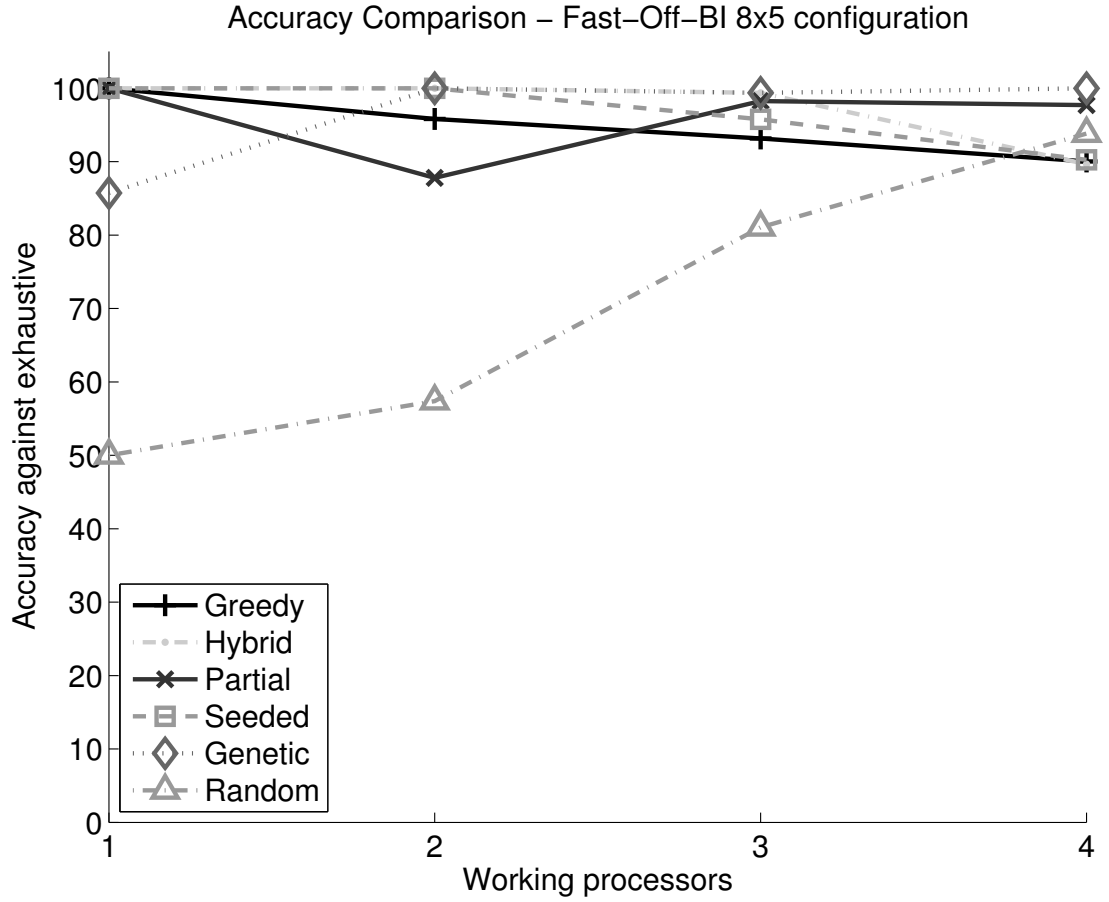


Figure 5.19: Solution accuracy comparison, Fast-Off-BI configuration

- *Hybrid* is one of the most accurate solutions (keeping almost always above 80%), which is to be expected given the fact that it inspects the largest fraction of the state space compared to other state-space traversing algorithms.
- *Partial* proves inconsistent, which can be largely attributed to the fact that it has a completely random starting point. However, it still maintains a higher degree of accuracy than *Greedy*.
- *Seeded* seems to **always** improve upon the quality & accuracy of both *Partial* and *Greedy*, which means that the idea behind its inception is a valid one.
- *Genetic* is the most accurate solution of all algorithms. It has proven to be just as accurate as an exhaustive search, at least for the input sizes that allowed for such a comparison. It is consistently the algorithm with the highest accuracy.
- *Random* has a low and inconsistent quality & accuracy, as was expected by the definition of the algorithm. It is worth mentioning that all non-deterministic algorithms proved to be consistently better than the purely random strategy.

5.2.5 Simulated multiprocessor array performance

Finally, we decided to make some measurements in the experimental setup that was described in section 4.3: Since we have the setup available to create any bubble-injected processor that we wanted, and we could also simulate with cycle-accuracy their actual performance, we could actually simulate a hypothetical 8x5 multiprocessor array with defects, as it would perform had it been reconfigured by one of our algorithms.

For this measurement, we chose $N = 8$ to have enough processors for the algorithm to reconfigure, and since the pipeline we extracted the Bubble Impact heuristic from in the simulation had $M = 5$, we used an 8x5 array as our hypothetical multiprocessor array.

We then employed the *Genetic* algorithm, which was the most accurate algorithm as shown before, and in a second experiment the *Greedy* algorithm, which was the fastest one, to solve our 8x5 defective array inputs. Since $N = 8$, the algorithms had to solve 7 different instances, with up to 7 working processors (as we explained, the case of 8 working processors means no defects). For *Genetic*, we used the “Accurate” tuning and the “Off” setting for constraints for this measurement to see the best results that we could get in terms of accuracy. For *Greedy* there are no parameters to set, so the only relevant option is the constraint usage, which was set to “Off” like with *Genetic*.

The evaluation procedure was as follows: We provided both algorithms with the same 7 inputs to solve. We then used the procedure described in section 4.3 to create each working processor that the algorithm provided in each problem instance. For example, when *Greedy* provided a solution with 5 different working processors (each with a certain bubble injection pattern), we created these 5 processors individually in Processor Designer with the dictated amount of bubbles injected. We then simulated the execution of the three micro-benchmarks defined in section 4.3 and measured how many cycles were needed for their completion. This was carried out for both algorithms in all 7 instances.

Since we know the initial amount of instructions in each micro-benchmark and how long (in cycles) a processor with no bubbles needed to execute each micro-benchmark, we can calculate the *Instructions Per Cycle* (IPC) for an ideal (baseline) processor. And since we can also simulate and measure how many cycles a bubble-injected processor needs to complete the same amount of instructions (same micro-benchmarks), we can calculate its IPC as well. Since we also define each processor in the array to be independent (in terms of instruction execution) from the others, we can simply add the IPCs of all the processors in an array to get the total IPC of the entire array.

It is worth mentioning again here that the Bubble Impact heuristic expresses processor performance in IPC, since it calculates an estimation of the reciprocal of the cycles that a processor needs to execute a known benchmark, as shown in the heuristic function’s formal definition in section 4.3. So since estimation and simulation produce quantities using the same measurement units (IPC), we can actually compare them to see exactly how close our estimations of (expected) performance cost were to the actual (simulated) performance cost. This is an important piece of information since we have stressed that heuristic accuracy is a determining factor in algorithm performance.

In Figure 5.20 we see a comparison of the IPC (y-axis) for each of the 7 instances (x-axis) that the two algorithms solved, normalized to 100% per working processor. At

each instance (each point on the x-axis) we show:

- The “ideal” IPC, which is the IPC that would have been obtained if we had x amount of ideal (100% IPC) processors
- The “estimated” IPC, which is the IPC that the Bubble Impact heuristic calculated for the array that the algorithm provided as an output
- The “actual” IPC for each benchmark, which is the IPC that we obtained for the array with the simulation procedure described above.
- The “gap” from the simulated IPC results to the ideal state, expressed as a percentage.

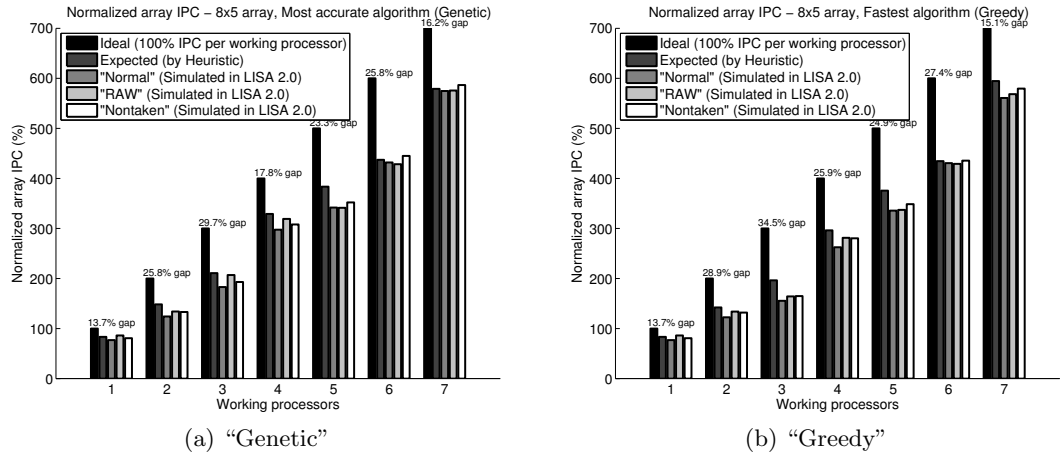


Figure 5.20: Heuristic accuracy: Bubble Impact IPC vs Simulated IPC

From it, we can make the following conclusions:

- As expected and mathematically proven in section 4.3, as the available working processors increase, so does the IPC.
- The heuristic is actually **very accurate** - The estimated IPC is consistently close to the actual IPCs obtained by simulation, for all instances in both algorithms. In fact, they are 93.72% accurate for the IPC's of *Genetic* and 91.75% for those of *Greedy*, on average.
- We also observe that, in general, the arrays produced tend to follow the ideal state rather closely with consistency (as indicated by the gap percentages shown on the figure).

Using the same numbers that we obtained from these experiments, we can make another useful comparison: How do the arrays obtained by the fastest algorithm perform, compared to the arrays provided by the most accurate algorithm. In order to do that comparison, we must view each micro-benchmark simulation separately since each of

them has a different amount of instructions. For this comparison we show (in Figure 5.21) again the IPCs of each of the 7 arrays provided by the two algorithms in each input instance, normalized to 100% IPC per working processor. At each instance (each point on the x-axis) we show the “ideal” IPC as we defined it previously, and the IPC of the array that *Greedy* and *Genetic* provided.

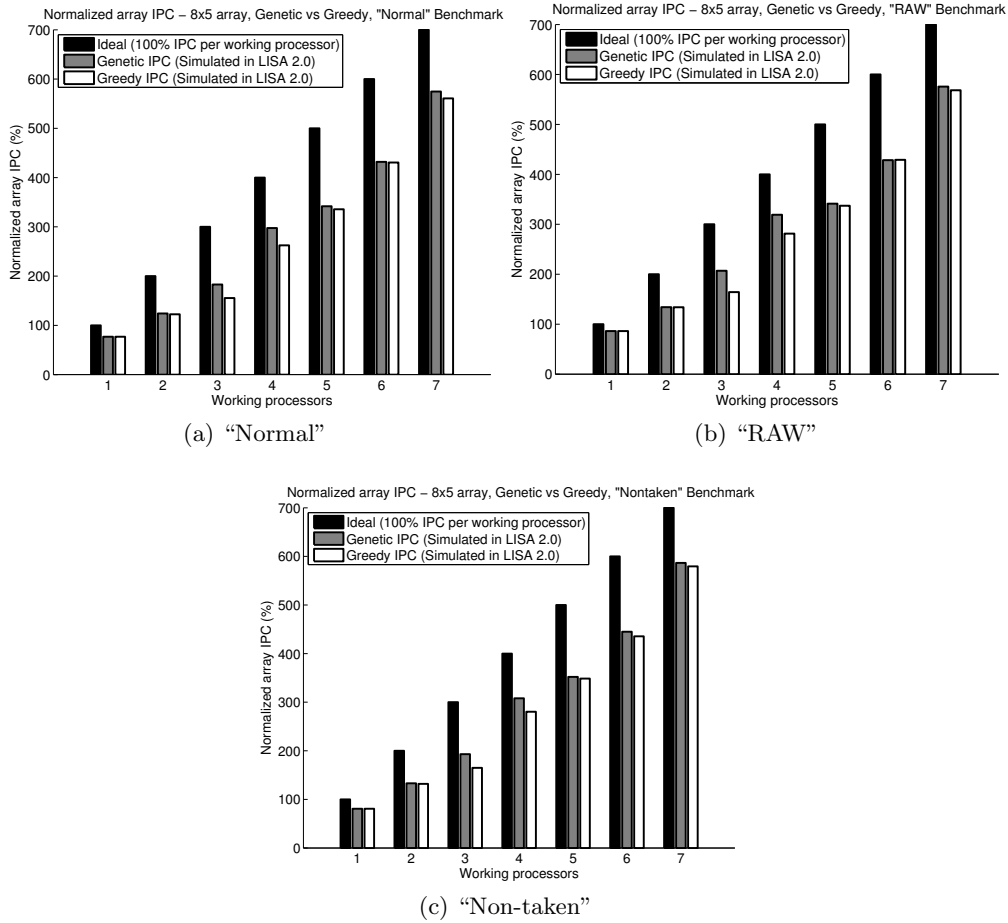


Figure 5.21: *Genetic* vs *Greedy* vs *Ideal* IPC comparison

From it we can see that *Genetic* is consistently better than *Greedy* as expected by the empirical evaluation results presented in the previous section. However, *Greedy* is not that much worse, performing somewhat better than expected (i.e. closer to the results of *Genetic* than expected) by the heuristic function as shown in the empirical evaluation before.

5.2.5.1 The “special” case of bursty processing demand

It can be argued that IPC measurements give a picture that is somewhat unrealistic/idealized, since it is a metric averaged over time. It could be that jobs come in bursts, with a lot of idle time between each burst. This could mean that even though

an array was reconfigured to have the maximum amount of processors possible, some of them could be slow enough, that, given a bursty job demand, would not contribute to the performance of the array at all. This can be the case for example when scheduling the jobs on processors that are faster than others finishes those jobs in a smaller amount of time than a more balanced job distribution.

To see the effects of this phenomenon, we decided to make the following calculations from the results obtained from the previous experiment: We assume that there are only as many jobs to be carried out as there were processors in the array before defects arose, i.e. the minimum amount possible that would utilize all the (initially) available processors. In our case, this means that we assume that only 8 jobs (8 executions of a micro-benchmark) were to be carried out on the processor array. Then, using the cycles that we measured before (each processor, of each array at each instance, for each algorithm) we can, by manually performing an optimal scheduling of these jobs, find out how many cycles in total would be needed to complete all 8 of them in each case. Again, since each micro-benchmark has a different amount of instructions, we need to do this comparison for each micro-benchmark individually.

The results of this comparison are shown in Figure 5.22, where we show the total amount of cycles per instance, for the two algorithms as well as the ideal cycles. In this case, ideal means the cycles that x ideal processors would need to carry out the 8 original jobs. From it, we can see the following:

- Naturally, by adding more processors the total cycles needed to complete 8 jobs decrease.
- Indeed, the effect of adding more processors in the case of bursty processor demand seems to follow a diminishing returns trend: The more we approach the state where there are no defects, the less impact each new processor has. Or, inversely, the first processors that are knocked out due to defects cost less for the overall performance penalty. The more processors that are lost however, the impact of losing an additional processor becomes greater.
- It can be that the reconfigured array needs no more than the ideal amount of cycles, exactly because the optimal scheduling determined that defect-free (ideal) processors (if any existed in the reconfigured array) were to be assigned more tasks than others.
- *Greedy* and *Genetic* perform almost the same, and sometimes *Greedy* proves even better than *Genetic*. This is because of the fact that *Genetic* is **asymptotically** better than *Greedy*. This means that, for small enough time periods such as the scheduling of only 8 jobs instead of constant processing demand, it can provide solutions that prove worse in the short term. As shown however previously, the same solution proves to be better in the long term.

5.3 Conclusions

In this chapter we described the experimental setup that was used to evaluate our approach presented in chapter 4. We defined exactly what we would measure and how,

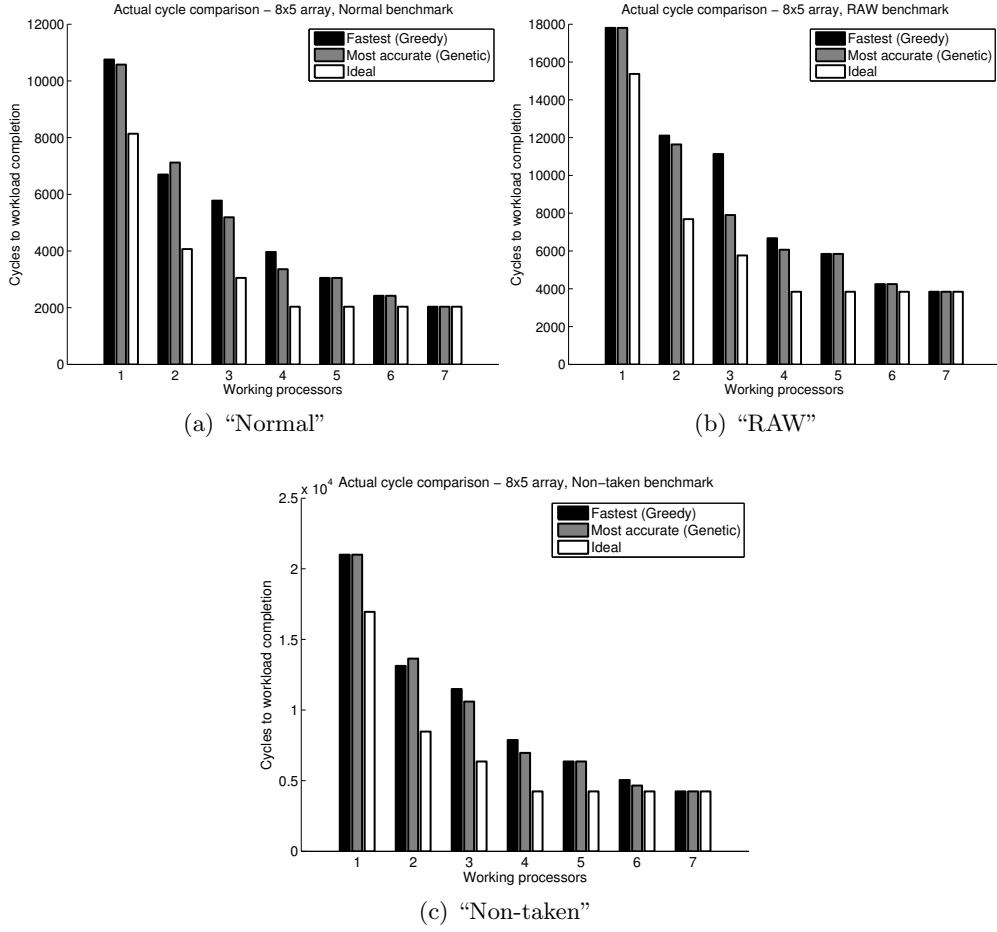


Figure 5.22: *Genetic* vs *Greedy* vs *Ideal* cycles (bursty demand)

and proceeded to present these measurements. We then also provided some conclusions based on the results that we obtained, for each metric that we evaluated. Additional discussion and further conclusions based on the evaluation results are given in the next chapter (chapter 6).

Conclusions

In this thesis, we attempted to approach the problem of exploring the vast amount of reconfiguration options in an adaptively defect tolerant multiprocessor array, and designing a way to produce an optimal or close to optimal solution to this problem. Challenges included the formal definition of the problem as a computational problem, defining performance metric approximations that would work well in as many scenarios of architectural designs and interconnect choices as possible, the exploration and addressing of the architectural issues that arise from an adaptive computer architecture for the purposes of simulation, the design of algorithms that can perform with reasonable time and space (memory) complexities and provide optimal or close to optimal solutions, and the empirical evaluation of our approach so that meaningful conclusions and observations can be made.

6.1 Conclusions on the evaluation results

Our results indicate that even though the reconfiguration options are many, the problem can be addressed by our algorithms and be solved in reasonable time. The fundamental trade-off that was detected from the problem definition between solution quality and time complexity was verified by the empirical evaluation of our algorithms, but we believe the cost in terms of time complexity to be worth paying in order to provide a better configuration.

During the empirical evaluation we saw the confirmation of many of the predictions made during the description of our approach:

- The problem indeed shows the typical behaviour of combinatorial explosion.
- The fundamental trade-off is as expected, between solution quality and time complexity. Space complexity (memory usage) remains within easily manageable limits.
- Different landscapes affect the performance of algorithms in different ways.
- We can indeed exploit the periodicity of the landscape to gain algorithmic speed, as shown by the empirical evaluation of *Partial*.
- *Greedy* will almost always provide a very fast solution but with the least accuracy.
- The solution provided by *Greedy* can be almost always improved by a few tweaks, as the quality evaluation of *Seeded* shows.
- The *Genetic* algorithm performs better than all other algorithms in terms of solution quality & accuracy, as expected from the fact that it is not hindered by the difficulty of various landscapes.

- A completely random search pattern as exhibited by *Random* will have worse performance than the other non-deterministic algorithms. In other words, clever non-deterministic methods are worth implementing instead of just leaving it all to chance.
- The Bubble Impact heuristic is a very accurate estimation of what the IPC performance of an array would be in practice.

The problem of algorithm selection has been studied and formulated as a computational problem in itself [21]. Given the trade-off between speed and accuracy based on the empirical evaluation we presented, the *Genetic* approach when solution quality is the critical factor is the most suitable approach. The *Greedy* strategy is more appropriate when speed is the most important factor, and when a fair trade-off between the two is sought after, the *Seeded* algorithm, or a faster tuning of the *Genetic* algorithm can be applied.

6.2 Thesis contributions

This thesis addressed a number of challenges regarding the design of adaptively defect tolerant multiprocessor arrays. The main contributions of this work include:

1. The proper definition of the problem as a formal computational problem in the combinatorial domain.
2. The definition of three general approximations for performance with varying degrees of architectural and interconnect knowledge integrated into these approximations.
3. The study and addressing of the architectural issues that arise from adaptively defect tolerant processors, for the purpose of simulation.
4. The design and implementation of eight heuristic search methods to address the issue of reconfiguration options.
5. The empirical evaluation of these implementations in order to reach useful conclusions.

Some key statistics from this work are:

- The problem size: 10^{19} candidate solutions at the largest input
- Optimal solution obtainable within 2.5 seconds, by the *Genetic* algorithm
- At least 70% of optimal within 10^{-4} seconds by the *Greedy* algorithm
- Cost estimations that are at least 91.75% accurate
- Solutions within 65% – 85% of the **ideal** state

We believe that the combination of defect tolerance and multiprocessor array design is a design paradigm that will become more prominent as technology scales, and that research in this field will draw increasingly more focus as we reach the limits of our current implementation technologies in terms of reliability and lifespan. This thesis is our contribution towards that direction, and we hope that it will prove to be a useful research resource within this field.

6.3 Future work

Some areas that can be further researched in this work include the investigation of other problem domains as a way to formulate this problem as a computational problem, as mentioned in section 4.1. It could be that a better choice in the trade-off lies within the linear programming domain, or the graph colouring domain, etc.

We also believe that the design choices for our algorithms can be further explored: Their parameters can take many different values that may impact their performance in ways that we did not expect. A systematic design exploration of the algorithm parameters in the context of this problem is a path that is worth investing research on. Perhaps even methods from Artificial Intelligence such as Machine Learning can be applied to finely tune the algorithms for specific demands.

The heuristic functions can always be improved to provide more accurate approximations. As we already mentioned, in the case of application specific processors, the heuristic function extracted from processor simulation could be carried out with the actual application code that the processors were designed for.

Of course, more algorithms could always be designed & implemented. The field of heuristic search provides a vast amount of options in respect to algorithm choice.

We also saw that scheduling policies have an impact on the performance of the array. There is extended research in the field of multiprocessor array scheduling, and methods from that field could be applied either to make heuristics more accurate or to extend the performance capabilities of a defect tolerant multiprocessor array.

Finally, it is worth investigating the architectural issues of a completely dynamic processor that can reconfigure itself on the fly when injected with variable amounts of bubble stages, as well as perhaps investigating the possibility to share stages between processors.

Bibliography

- [1] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith, *Configurable isolation: building high availability systems with commodity multi-core processors*, Proceedings of the 34th annual international symposium on Computer architecture (New York, NY, USA), ISCA '07, ACM, 2007, pp. 470–481. [26](#)
- [2] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao, *Reliable systems on unreliable fabrics*, Design Test of Computers, IEEE **25** (2008), no. 4, 322–332. [27](#)
- [3] D. Bhaduri, S. K. Shukla, P. Graham, and M. Gokhale, *Reliability analysis of fault-tolerant reconfigurable architectures*, IEEE Int. workshop on Design & Test of Defect-Tolerant Nanoscale Archit. (NANOARCH), May 2005. [27](#)
- [4] S. Borkar, *Designing reliable systems from unreliable components: the challenges of transistor variability and degradation*, Micro, IEEE **25** (2005), no. 6, 10–16. [ix](#), [1](#), [8](#), [9](#)
- [5] Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin, *Tolerating hard faults in microprocessor array structures*, Proceedings of the 2004 International Conference on Dependable Systems and Networks (Washington, DC, USA), IEEE Computer Society, 2004, pp. 51–. [27](#)
- [6] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky, *Bulletproof: A defect-tolerant cmp switch architecture*, In Proceedings of the 12th International Symposium on High Performance Computer Architecture, 2006, pp. 3–14. [27](#)
- [7] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to algorithms*, 2nd ed., McGraw-Hill Science/Engineering/Math, 2003. [46](#)
- [8] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271, 10.1007/BF01386390. [12](#)
- [9] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, CA, 1979. [32](#)
- [10] Brian T. Gold, Babak Falsafi, and James C. Hoe, *Chip-level redundancy in distributed shared-memory multiprocessors*, Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing (Washington, DC, USA), PRDC '09, IEEE Computer Society, 2009, pp. 195–201. [26](#)
- [11] David E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. [17](#), [58](#)

- [12] Vincent Granville, Mirko Krivnek, and Jean-Paul Rasson, *Simulated annealing: A proof of convergence.*, IEEE Trans. Pattern Anal. Mach. Intell. **16** (1994), no. 6, 652–656. [16](#)
- [13] S. Gupta, Shuguang Feng, A. Ansari, J. Blome, and S. Mahlke, *The stagenet fabric for constructing resilient multicore systems*, IEEE/ACM Int. Symp. on Microarchitecture (MICRO-41), nov. 2008, pp. 141 –151. [26](#)
- [14] S. Gupta, Shuguang Feng, A. Ansari, and S. Mahlke, *Stagenet: A reconfigurable fabric for constructing dependable cmps*, Computers, IEEE Transactions on **60** (2011), no. 1, 5 –19. [26](#)
- [15] Shantanu Gupta, Shuguang Feng, Jason Blome, and Scott Mahlke, *Stagenet: A reconfigurable cmp fabric for resilient systems*, Reconfigurable and Adaptive Architecture Workshop (RAAW), dec. 2007. [22](#), [26](#), [27](#)
- [16] Scott Hauck and Andre DeHon, *Reconfigurable computing: The theory and practice of fpga-based computation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. [1](#), [8](#), [9](#)
- [17] John L. Hennessy and David A. Patterson, *Computer architecture - a quantitative approach (4. ed.)*, 2007. [5](#), [6](#), [7](#), [35](#), [38](#), [41](#)
- [18] Holger H. Hoos and Thomas Stuetzle, *Stochastic local search: Foundations and applications*, 2004. [16](#)
- [19] B. Skaggs J. Emmert, C. Stroud and M. Abramovici, *Dynamic fault tolerance in fpgas via partial reconfiguration*, IEEE Symp. on Field-Programmable Custom Computing Machines, 2000. [27](#)
- [20] C. E. Stroud J. M. Emmert and M. Abramovici, *Online fault tolerance for fpga logic blocks*, IEEE Trans. Very Large Scale Integr. Syst. **15** (2007), 216–226. [27](#)
- [21] J. R. Rice, *The algorithm selection problem*, Advances in Computers **15** (1976), 65–118. [88](#)
- [22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), no. 4598, 671–680. [16](#)
- [23] Donald E. Knuth, *The art of computer programming, volume 4a: Combinatorial algorithms, part 1*, Addison-Wesley Professional, January 2011. [9](#), [10](#), [11](#)
- [24] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee, *Architectural core salvaging in a multi-core processor for hard-error tolerance*, Proceedings of the 36th annual international symposium on Computer architecture (New York, NY, USA), ISCA '09, ACM, 2009, pp. 93–104. [26](#)
- [25] Alex Ramirez, Felipe Cabarcas, Ben Juurlink, Mauricio Alvarez Mesa, Friman Sanchez, Arnaldo Azevedo, Cor Meenderinck, Catalin Ciobanu, Sebastian Isaza, and Georgi Gaydadjiev, *The sarc architecture*, IEEE Micro **30** (2010), 16–29. [1](#)

- [26] J. Ramos, J. Samson, D. Lupia, I. Troxel, R. Subramaniam, G. Cieslewski A. Jacobs, J. Greco, J. Curreri, M. Fischer, E. Grobelny, A. Georgr, and R. Some, *High-performance, dependable multiprocessor*, IEEE Aerospace Conference, 2006. [27](#)
- [27] T. Riordan, G.P. Grewal, S. Hsu, J. Kinsel, J. Libby, R. March, M. Mills, P. Ries, and R. Scofield, *The mips m2000 system*, Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on, oct 1988, pp. 366 –369. [7](#)
- [28] Bogdan F. Romanescu and Daniel J. Sorin, *Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults*, Proceedings of the 17th international conference on Parallel architectures and compilation techniques (NY, USA), PACT '08, 2008, pp. 43–51. [22](#), [27](#)
- [29] Kenneth H. Rosen, *Discrete mathematics and its applications*, 5th ed., McGraw-Hill Higher Education, 2002. [10](#), [11](#)
- [30] G.N. Gaydadjiev S. Tzilis, I. Sourdis, *Fine-grain fault diagnosis for fpga logic blocks*, Int. Conf. on Field-Programmable Technology (FPT 2010), Dec 2010, Beijing, China. [27](#)
- [31] Premkishore Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, *Exploiting microarchitectural redundancy for defect tolerance*, 21st Int. Conf on Computer Design, oct. 2003, pp. 481 – 488. [27](#)
- [32] Jr. Shuler, Robert, *Fpgas with reconfigurable fault-tolerant redundancy*, NASA Tech briefs MSC-24464-1, NASA Center: Johnson Space Center, 2010. [27](#)
- [33] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe, *Reunion: Complexity-effective multicore redundancy*, Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), MICRO 39, IEEE Computer Society, 2006, pp. 223–234. [26](#)
- [34] Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2 ed., Prentice Hall, 2003. [12](#), [15](#), [16](#)
- [35] D. Sylvester, D. Blaauw, and E. Karl, *Elastic: An adaptive self-healing architecture for unpredictable silicon*, Design Test of Computers, IEEE **23** (2006), no. 6, 484–490. [26](#)
- [36] V. Černý, *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*, Journal of Optimization Theory and Applications **45** (1985), no. 1, 41–51. [16](#)
- [37] I. Wagner, V. Bertacco, and T. Austin, *Shielding against design flaws with field repairable control logic*, Design Automation Conference, 2006 43rd ACM/IEEE, 0-0 2006, pp. 344 –347. [27](#)

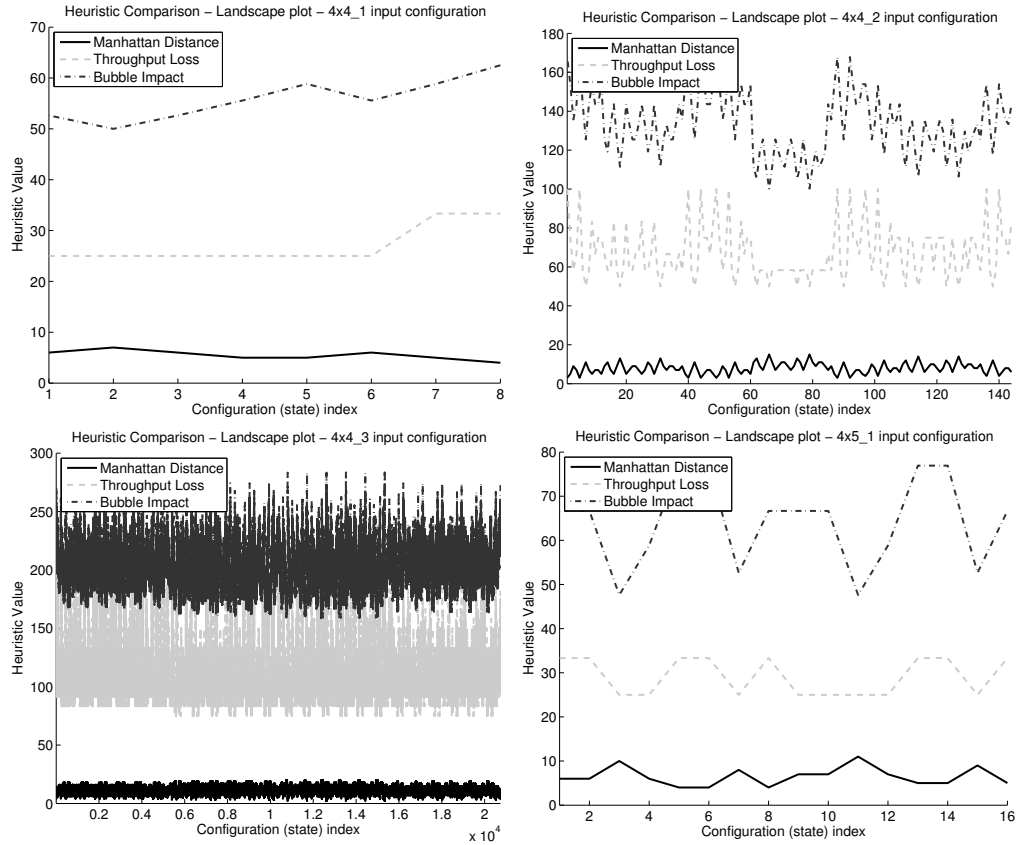
Appendix: Extensive Evaluation



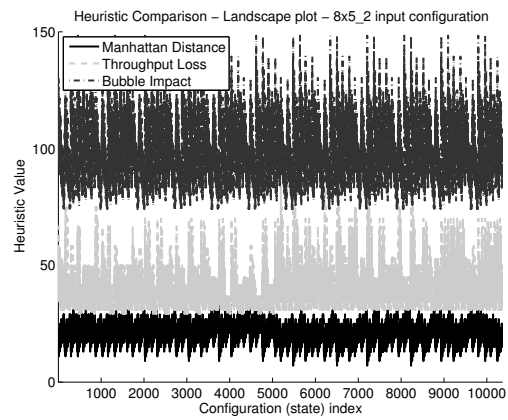
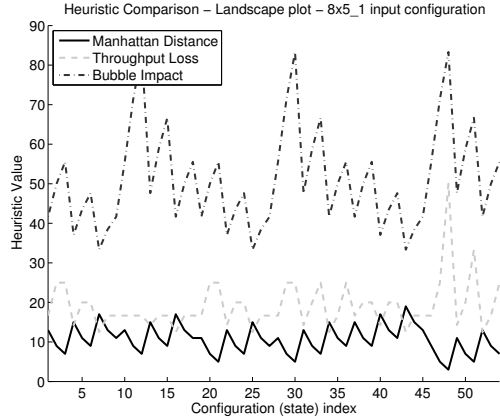
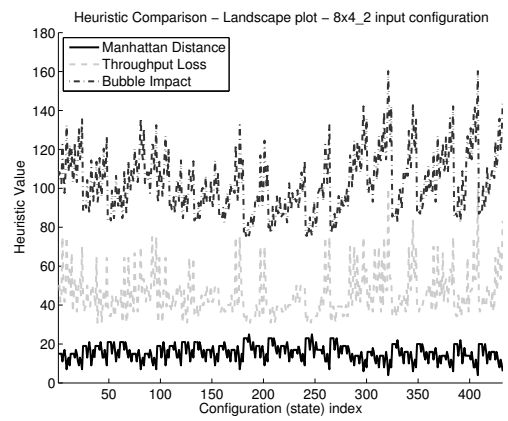
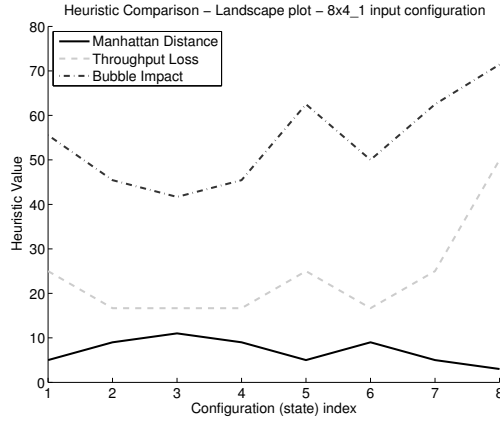
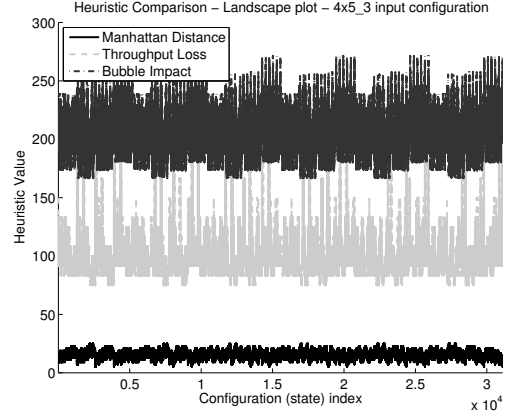
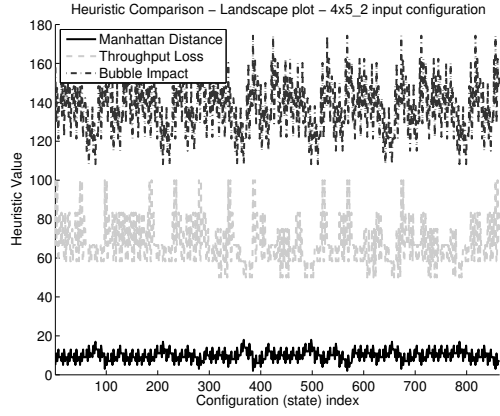
Here we present the extensive evaluation results, for every array size, every benchmark configuration and every measurement. They appear in the following order:

1. Landscape plots¹
2. Execution time plots
3. Solution quality plots
4. Solution accuracy plots

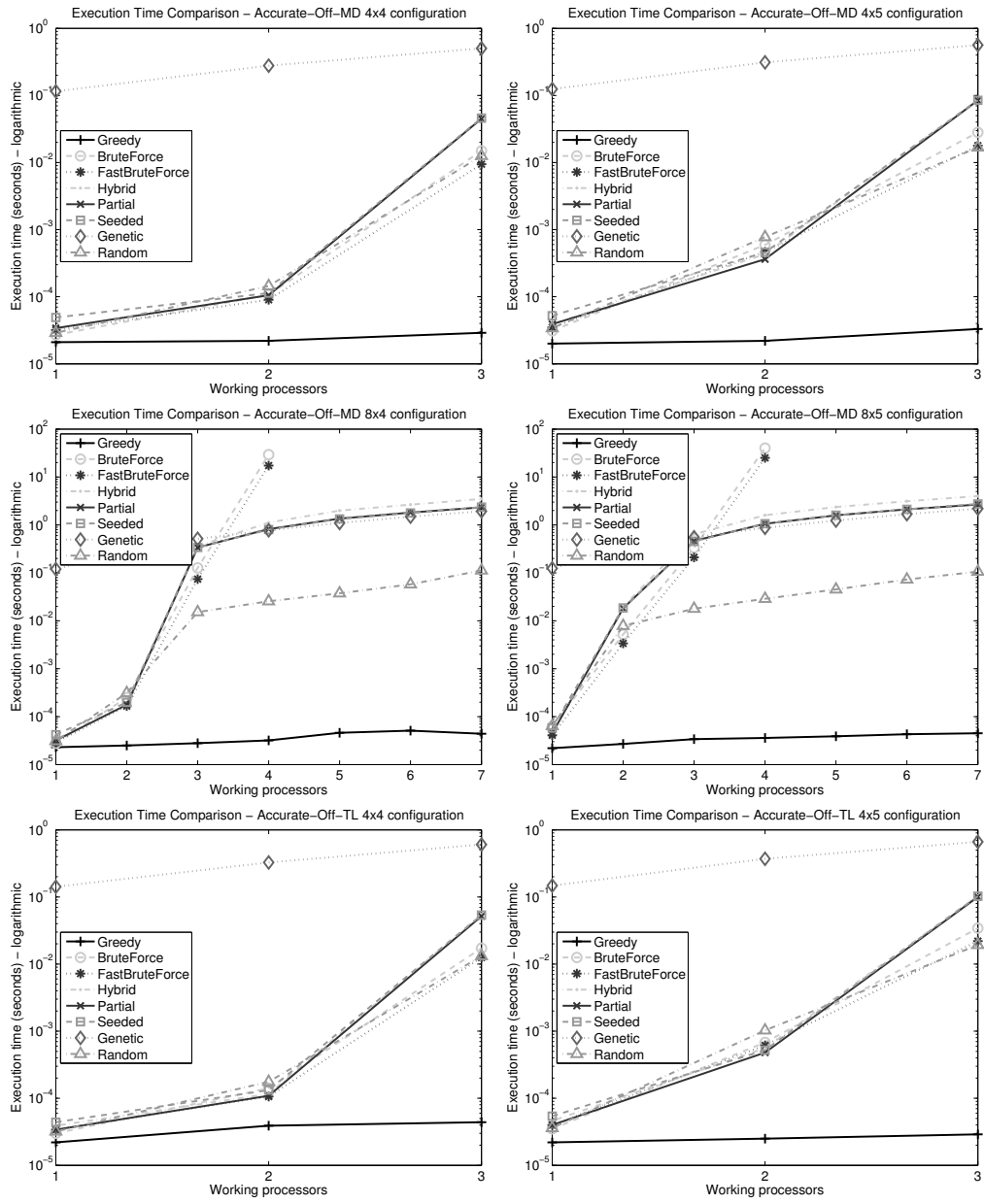
A.1 Landscape plots

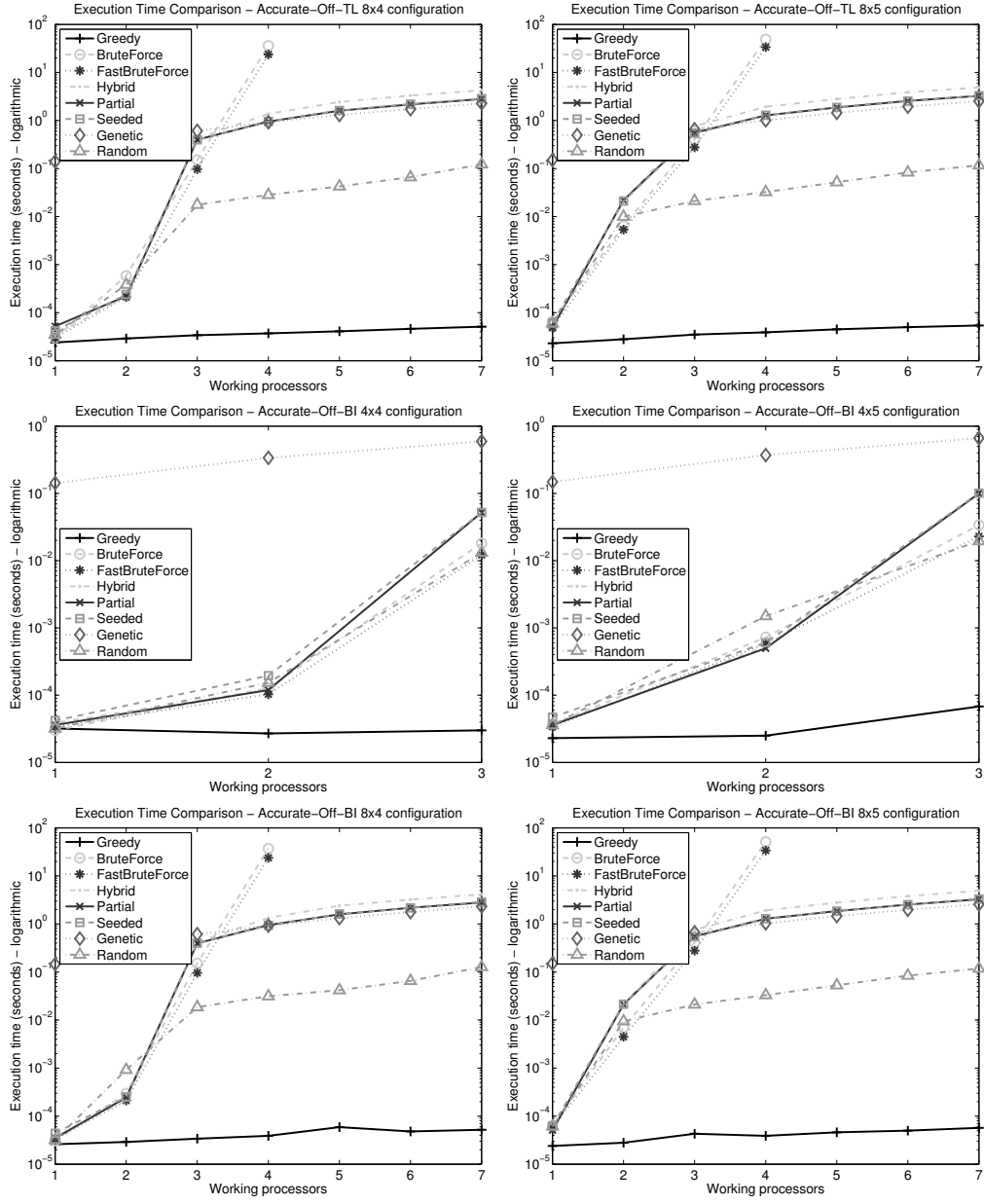


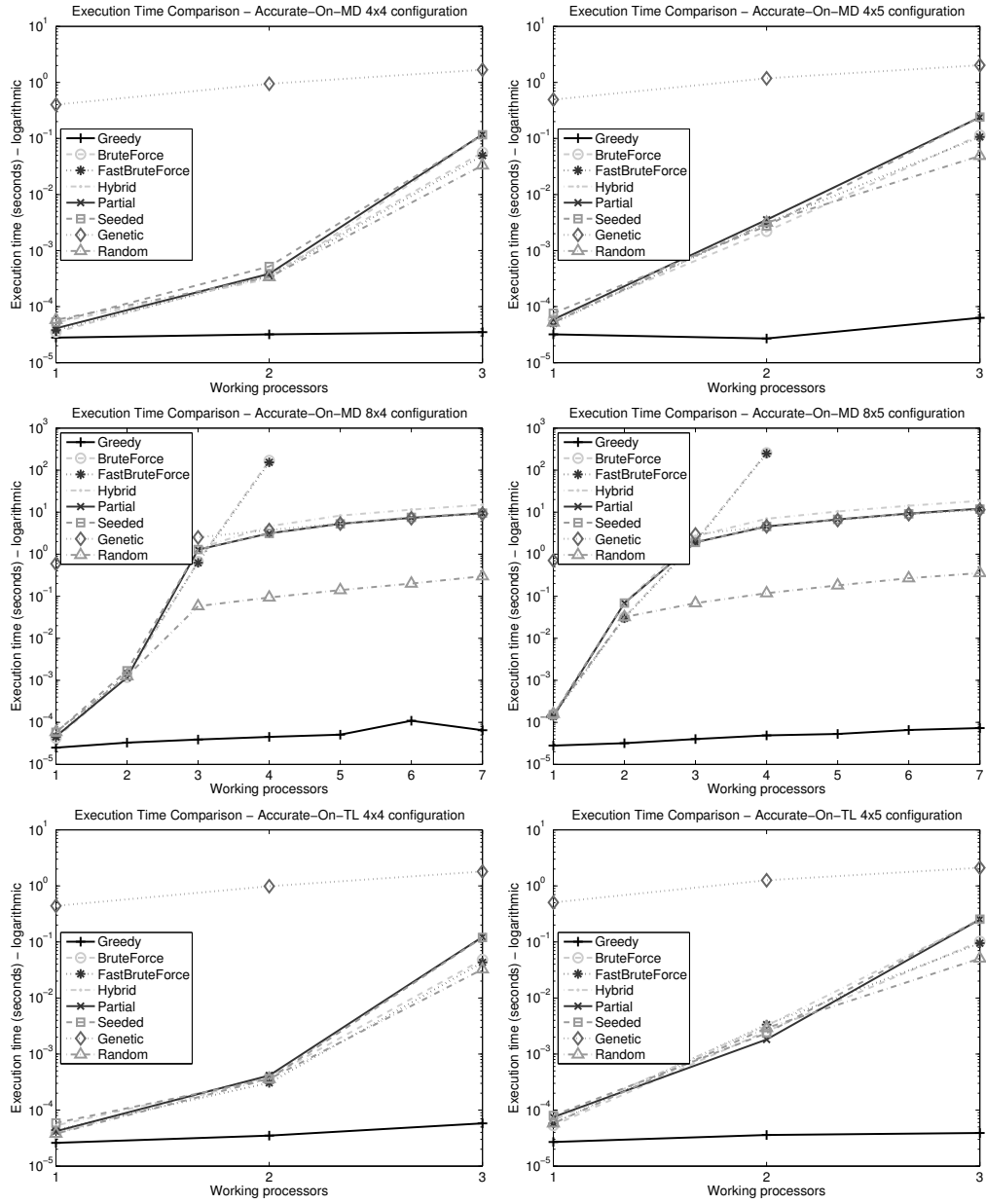
¹Plots for input sizes 8x5 and 8x4 are shown until $k = 2$ for readability's sake

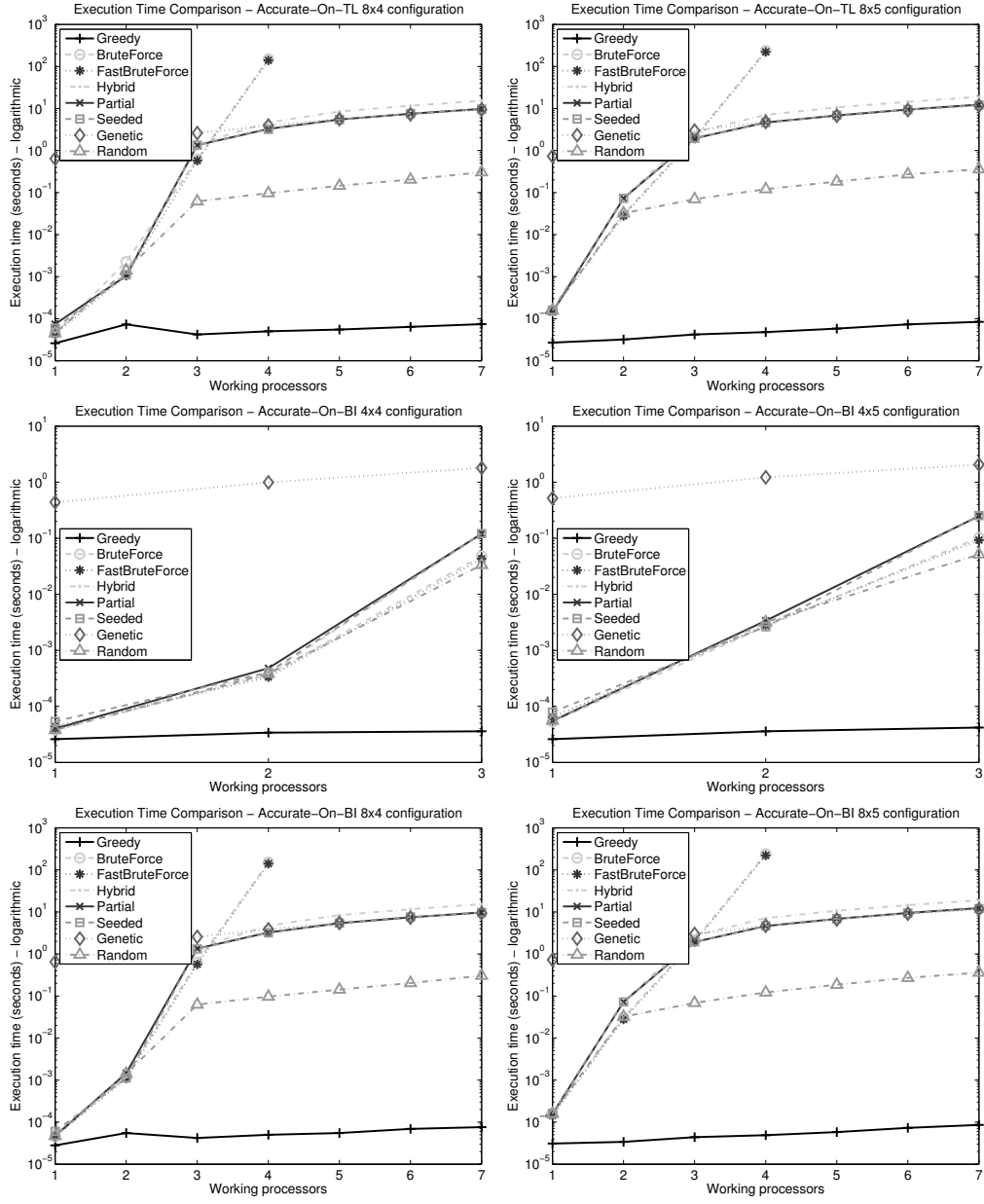


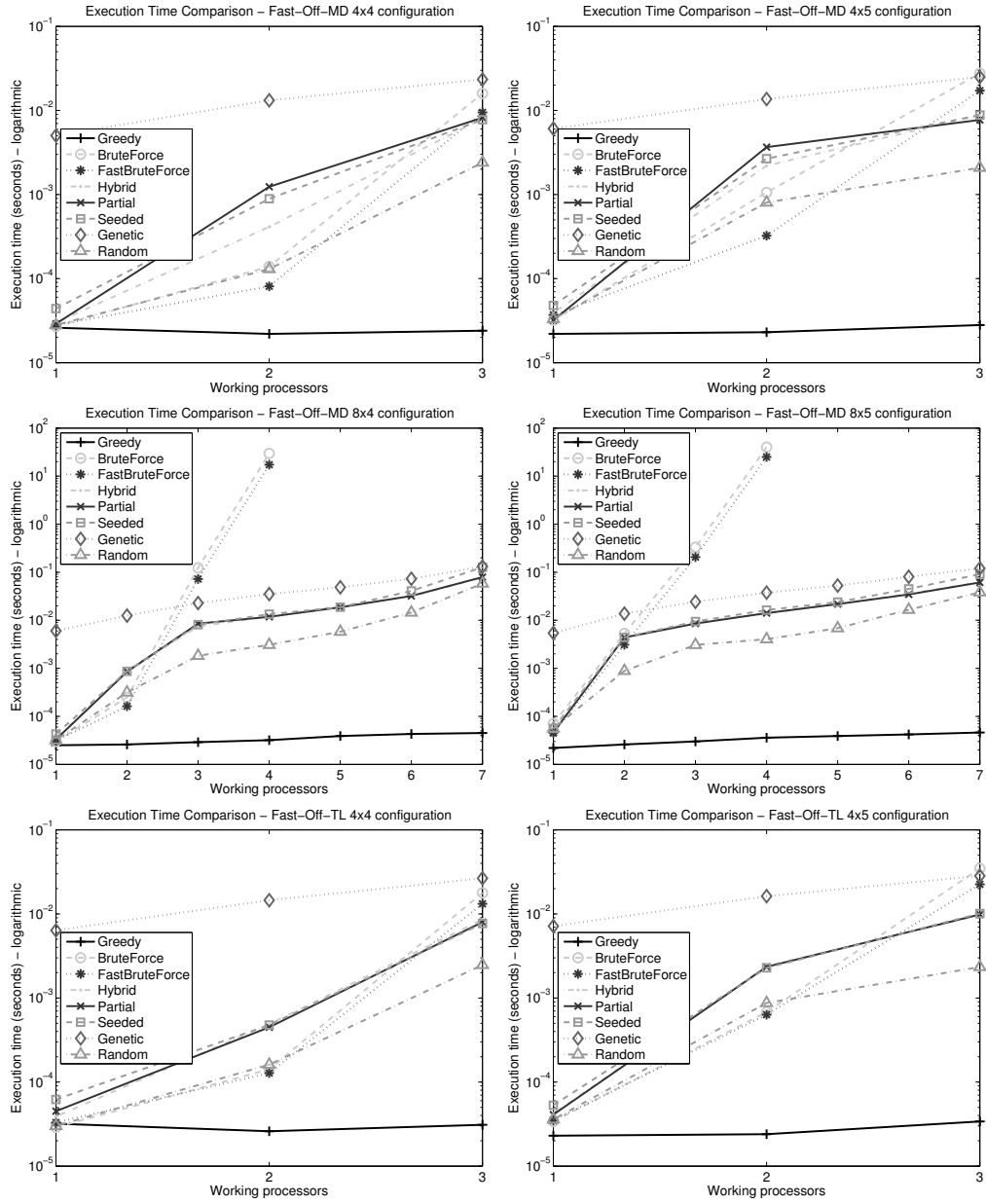
A.2 Execution time plots

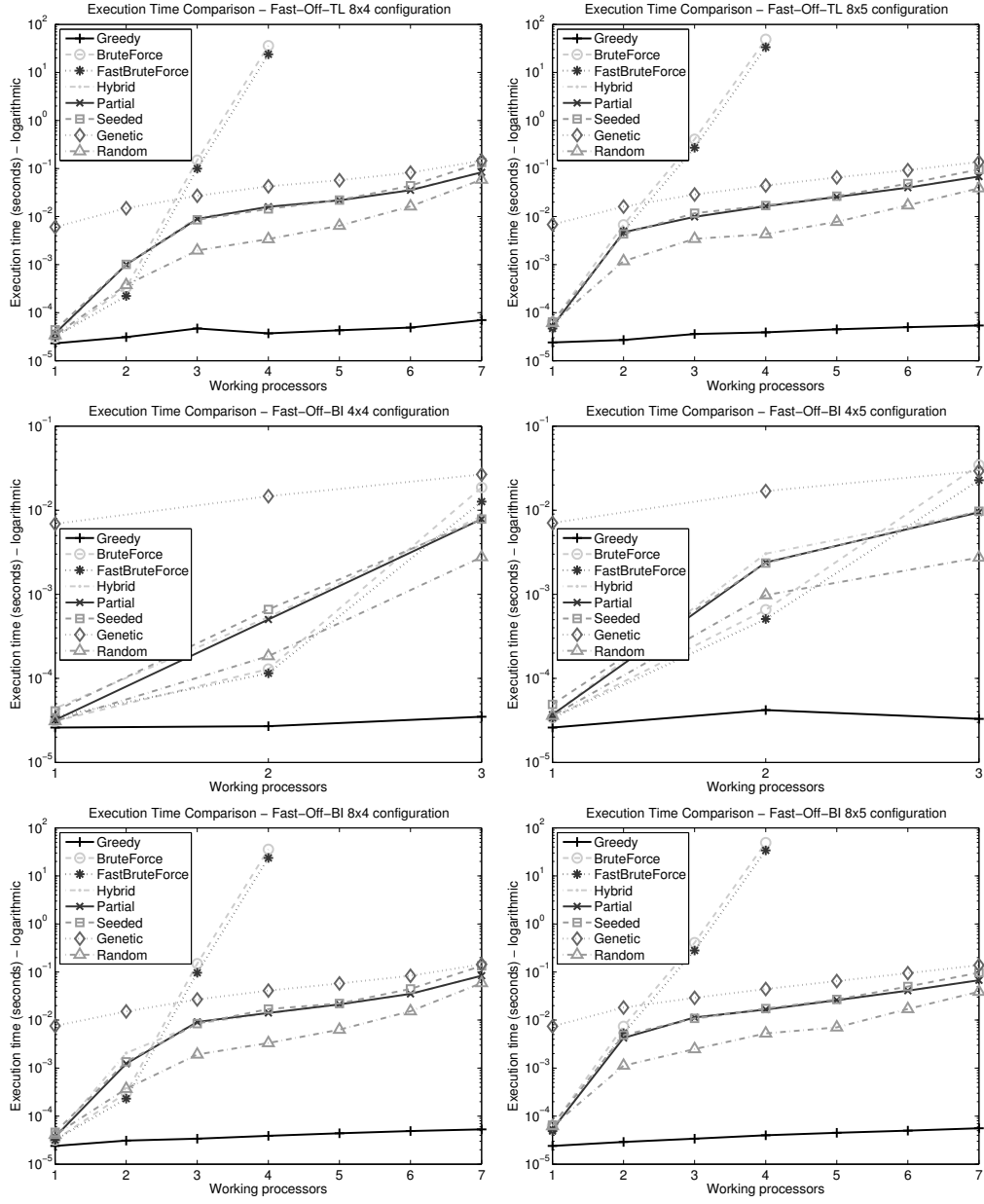


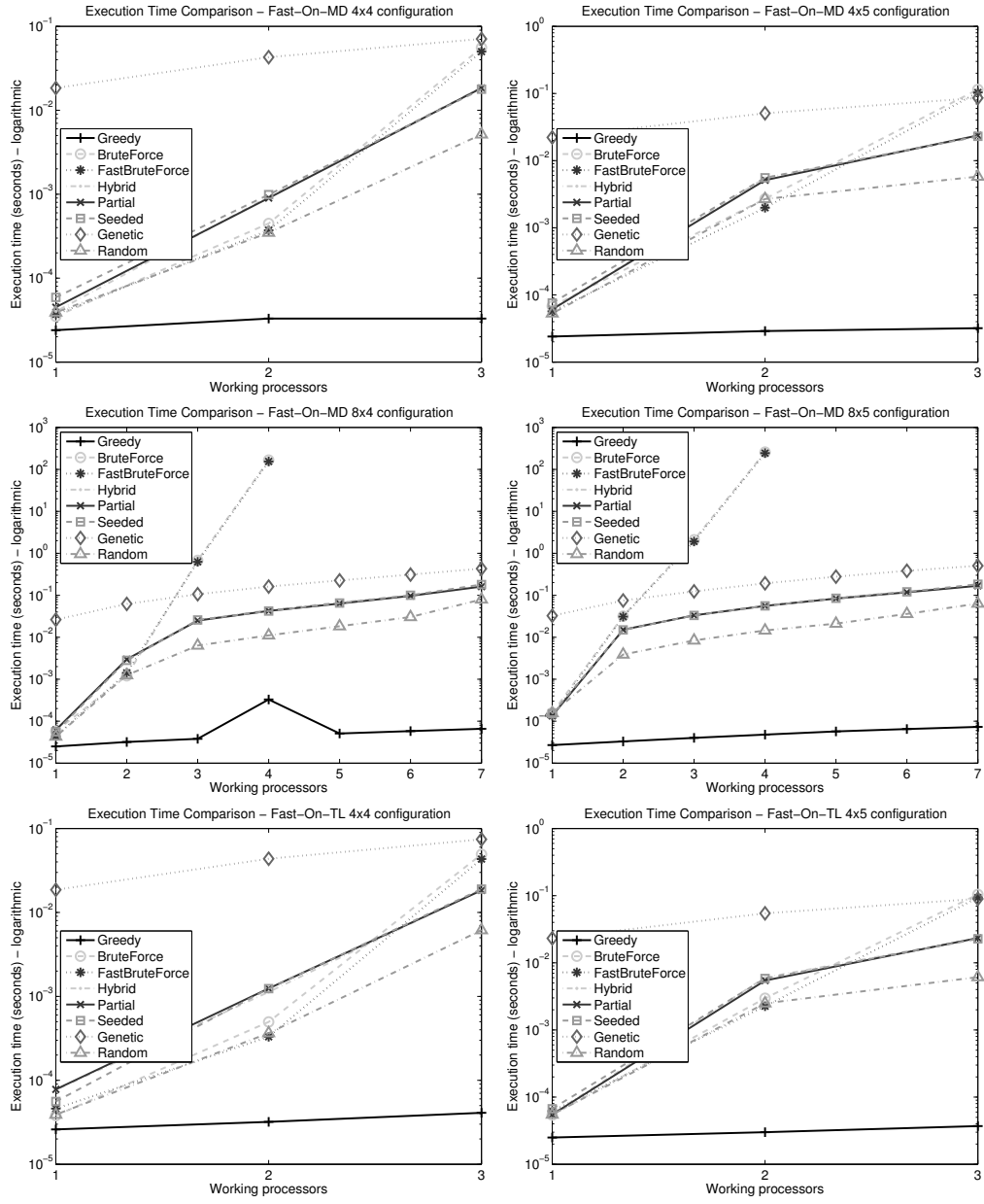


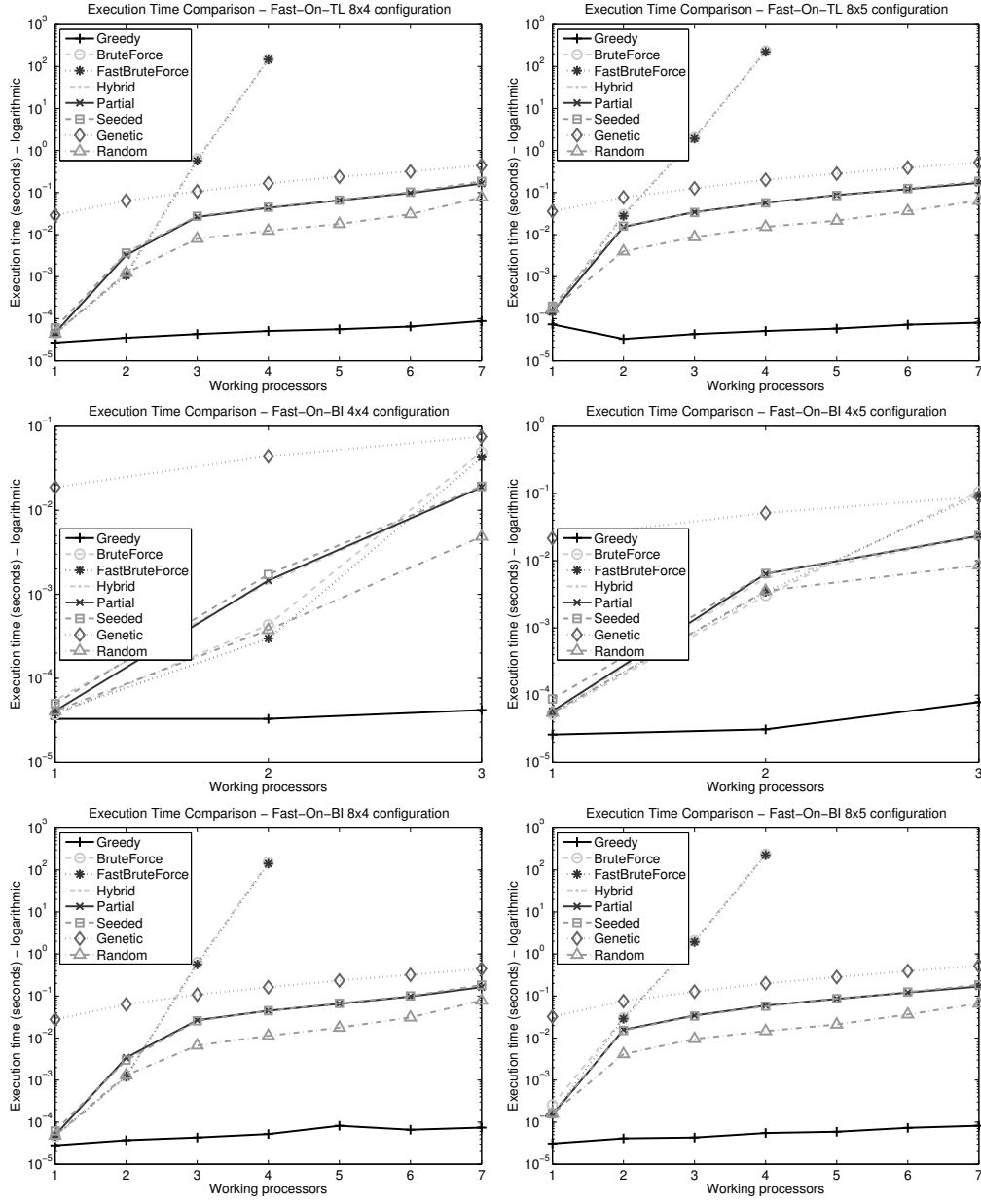




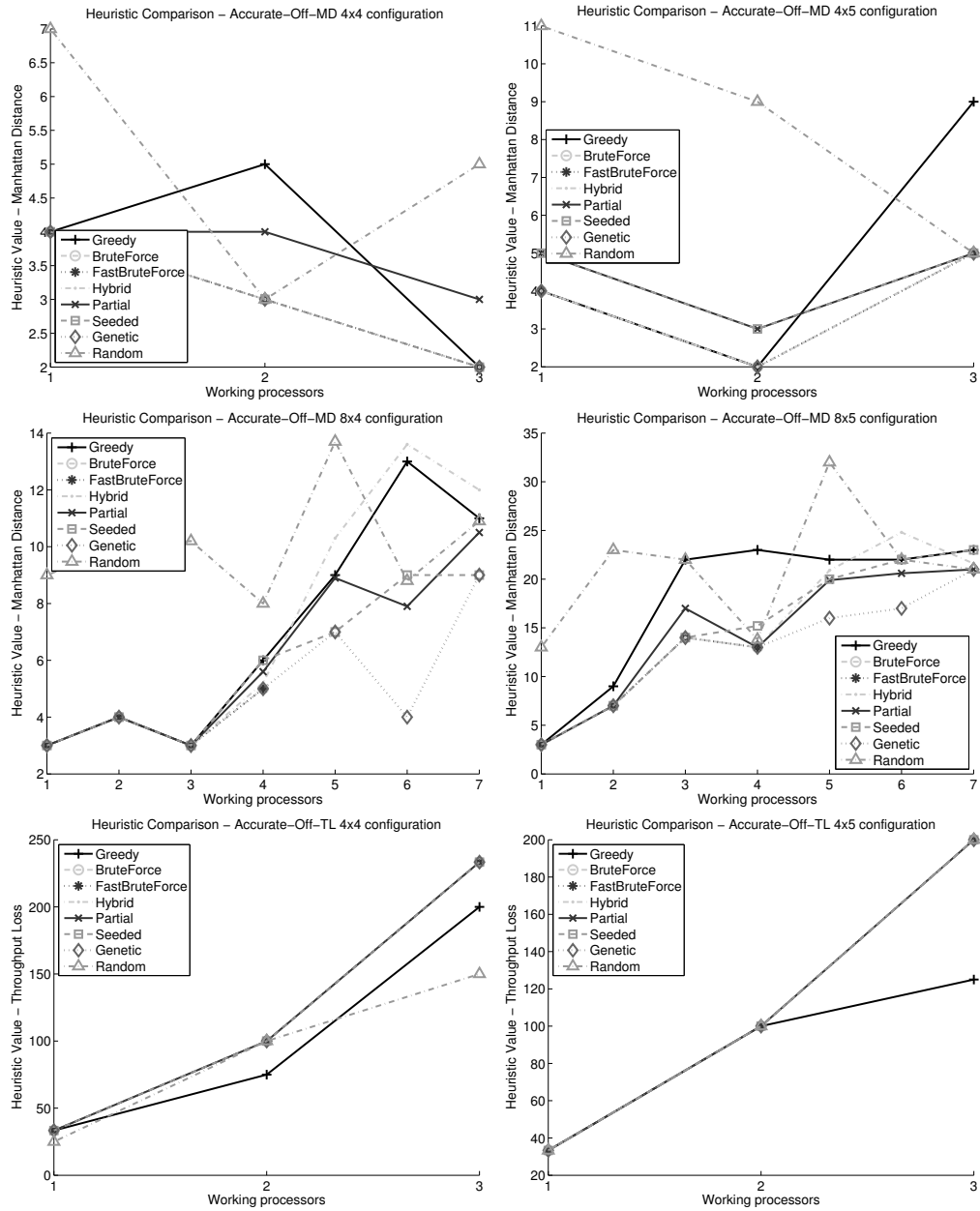


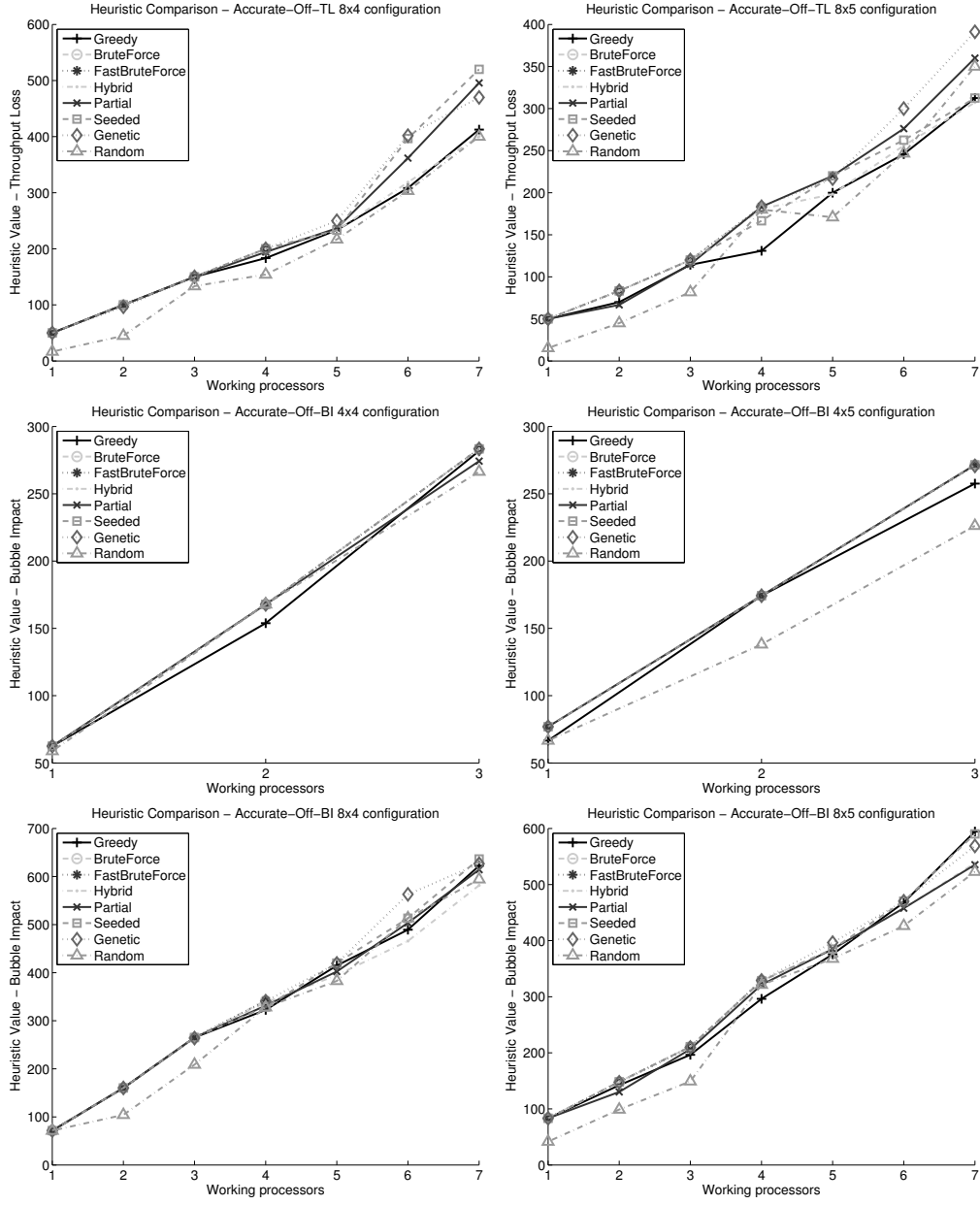


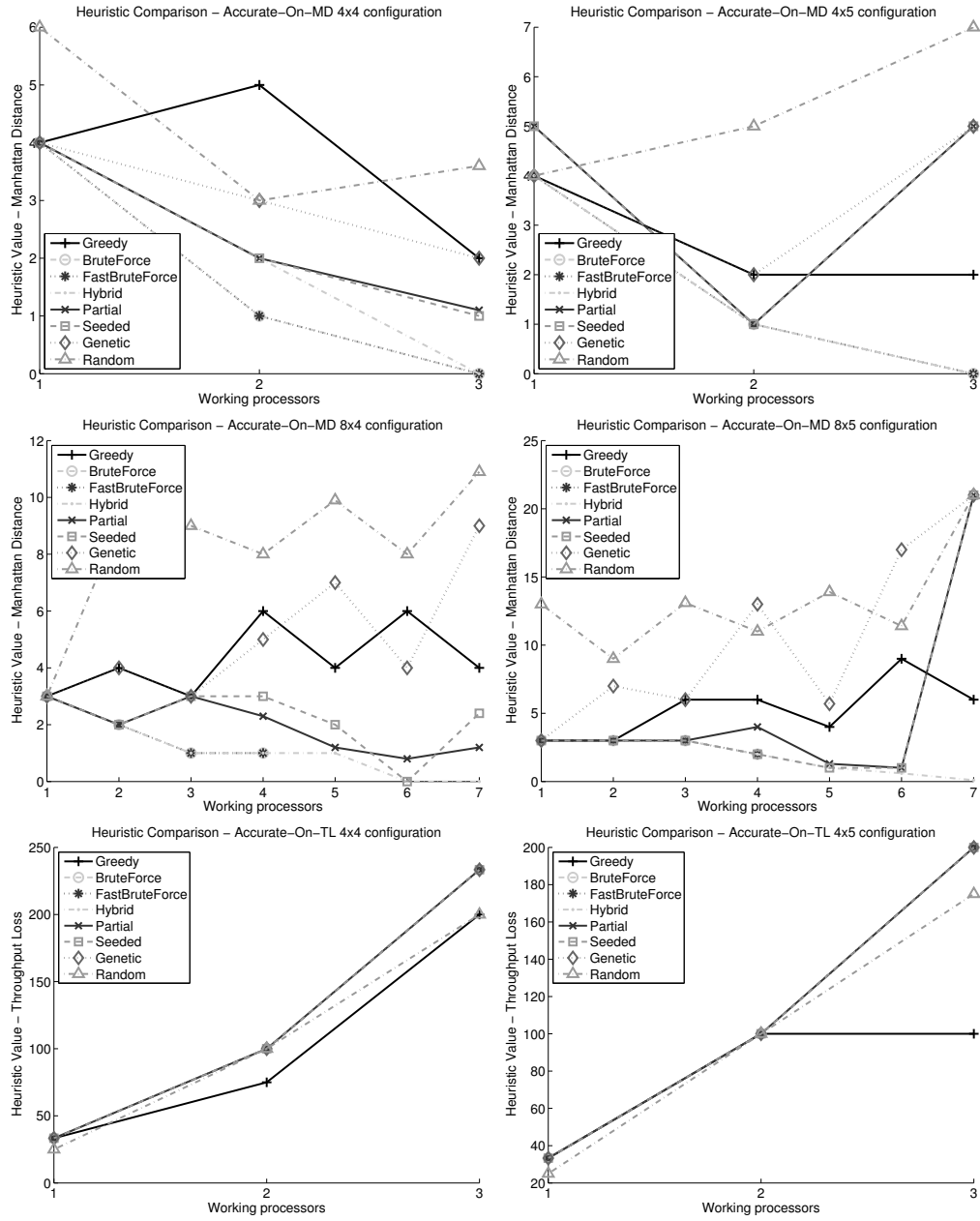


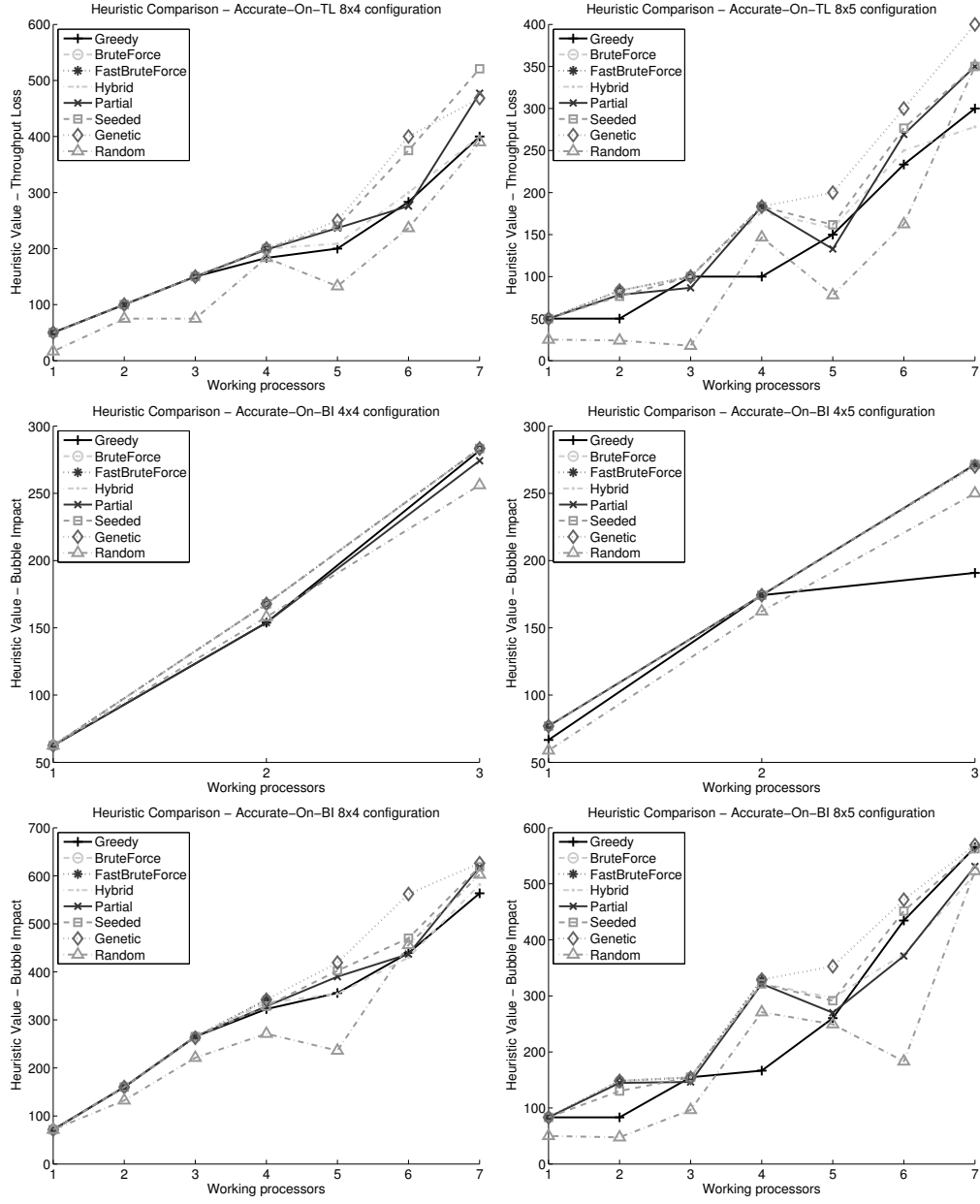


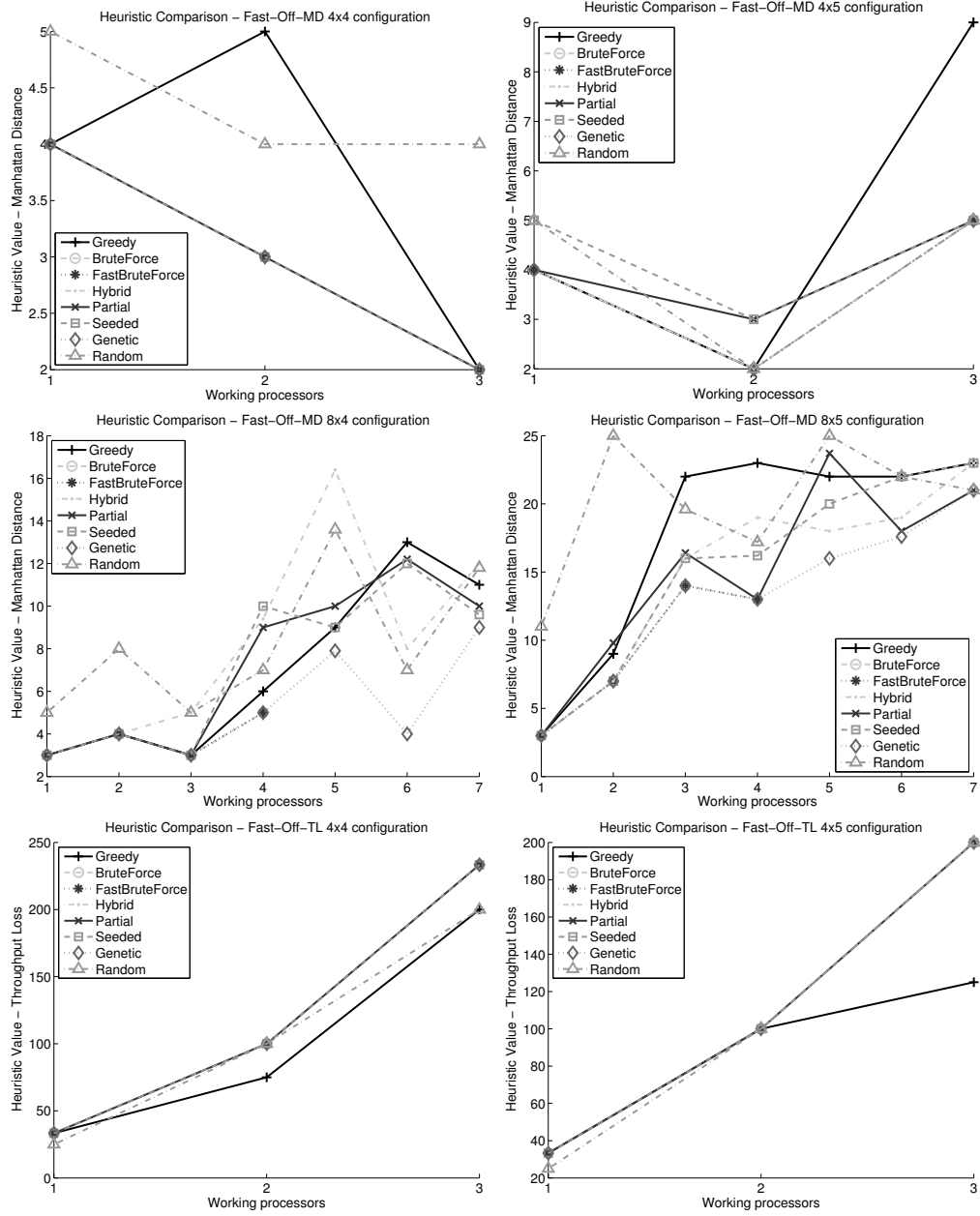
A.3 Solution quality plots

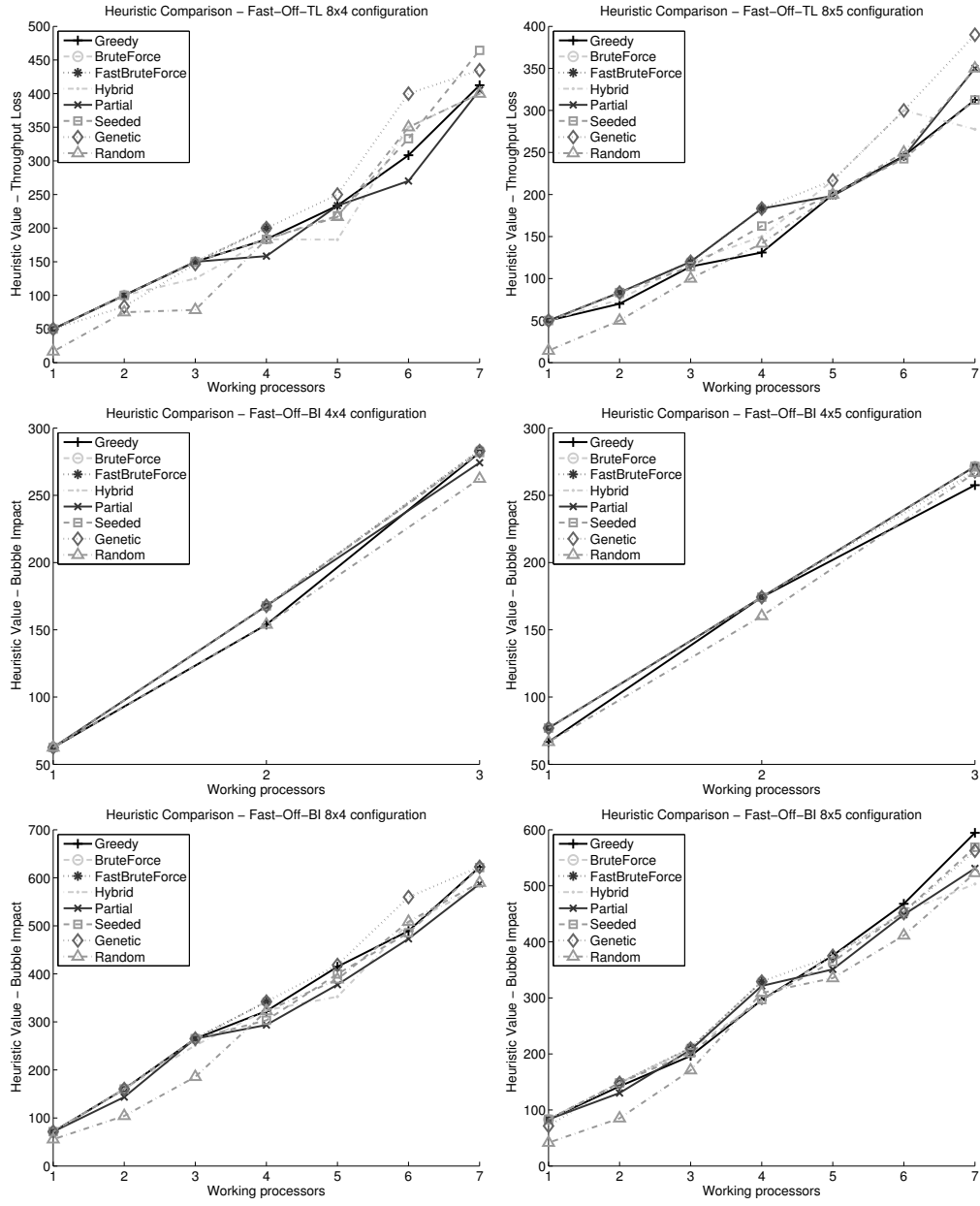


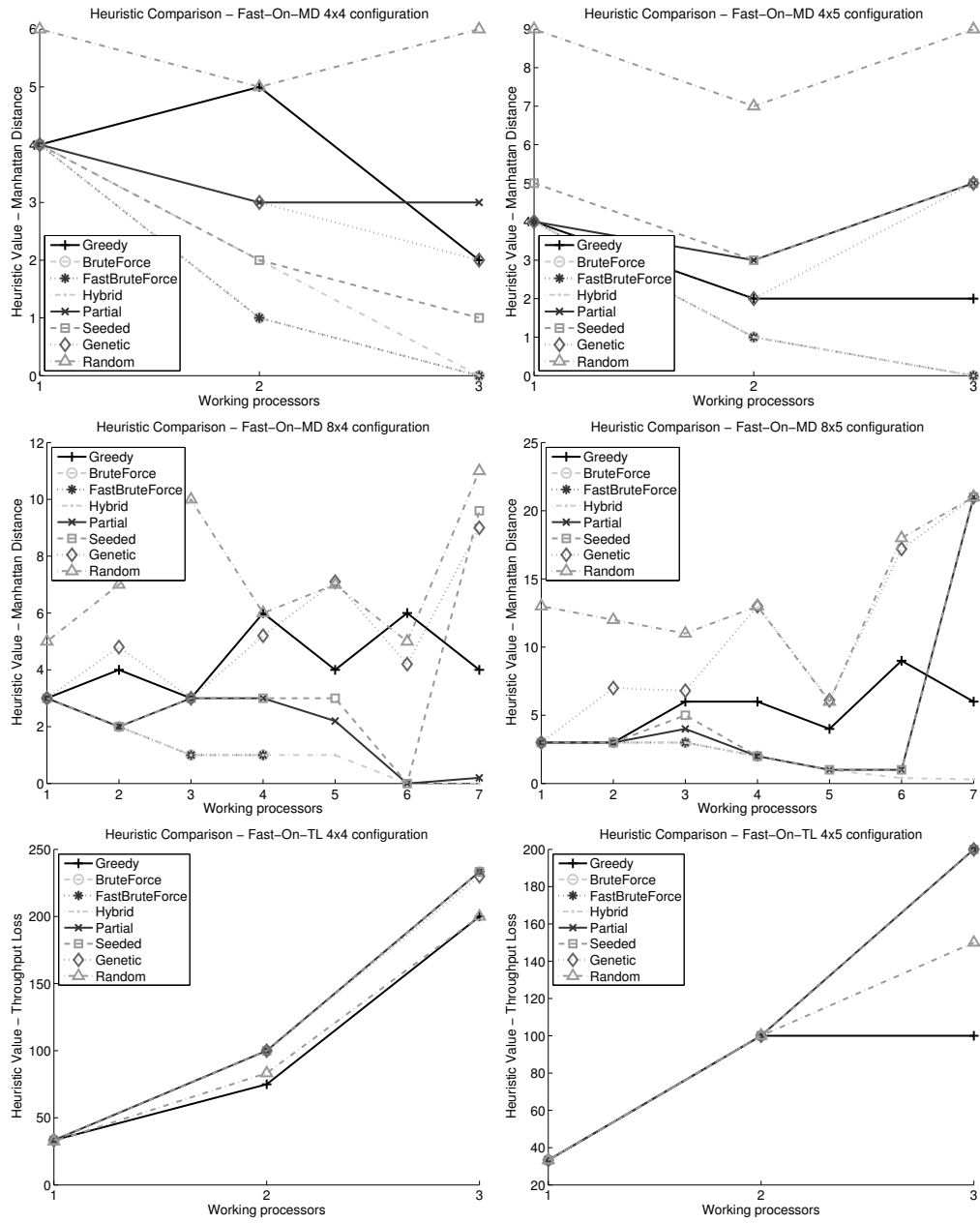


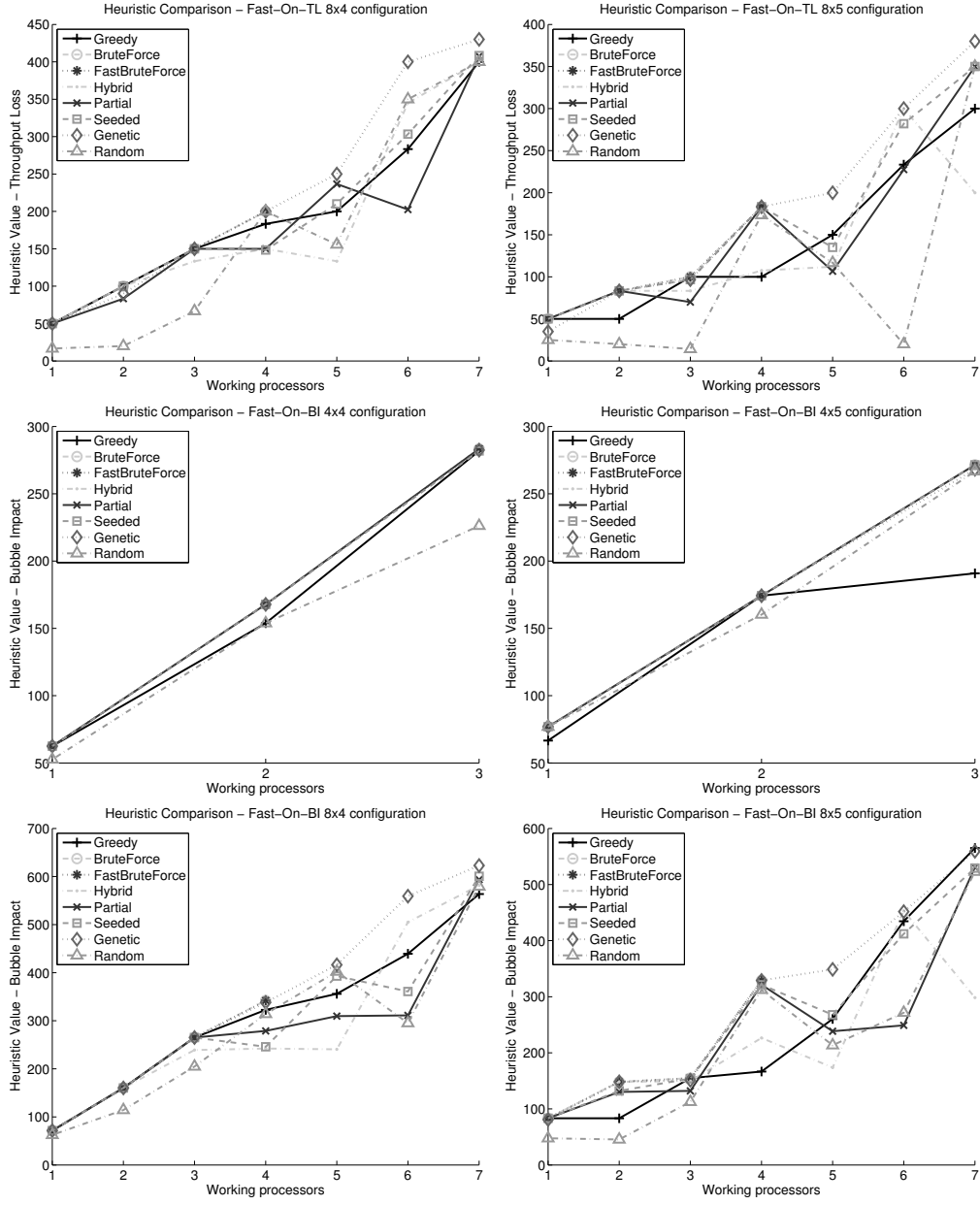




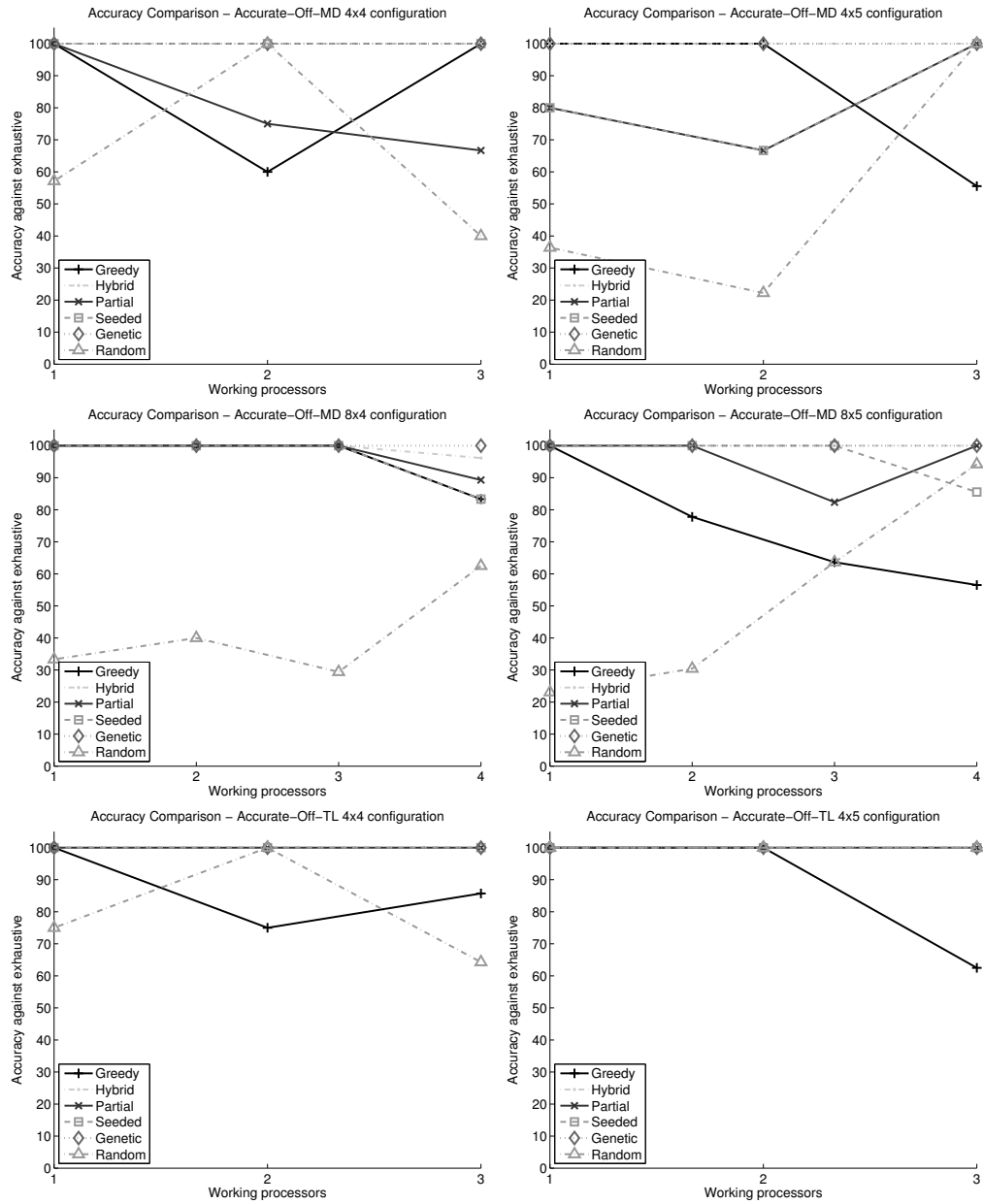


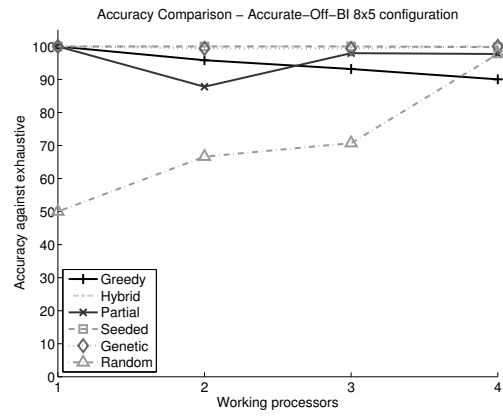
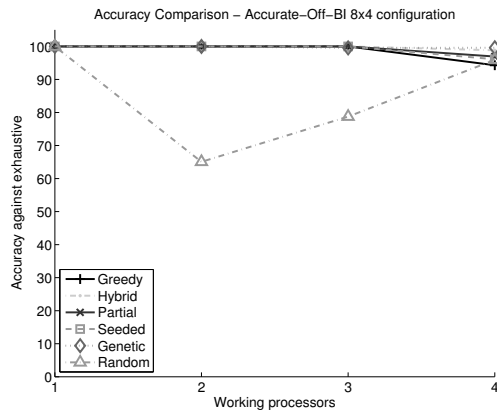
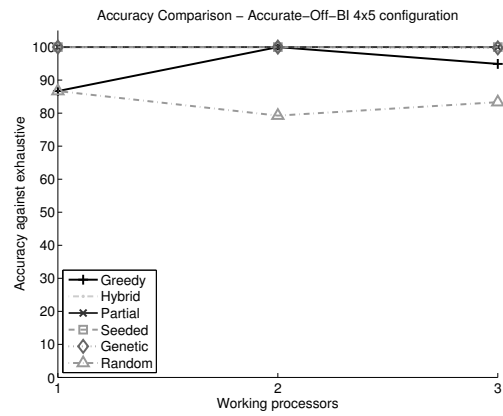
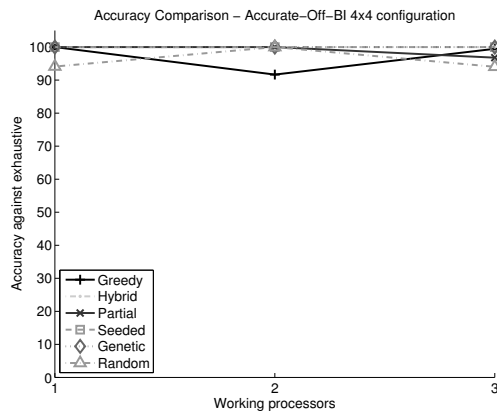
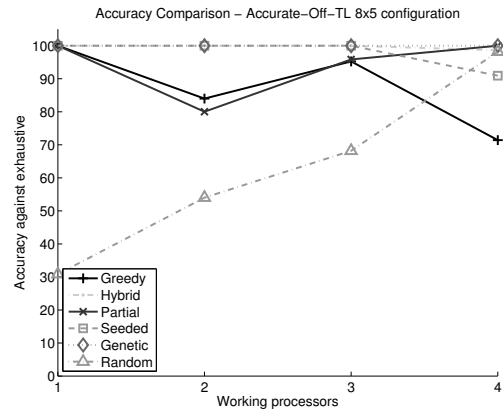
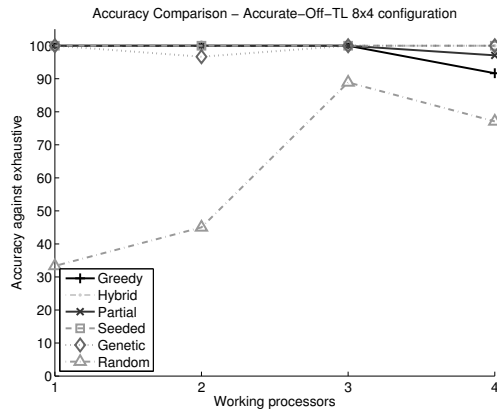


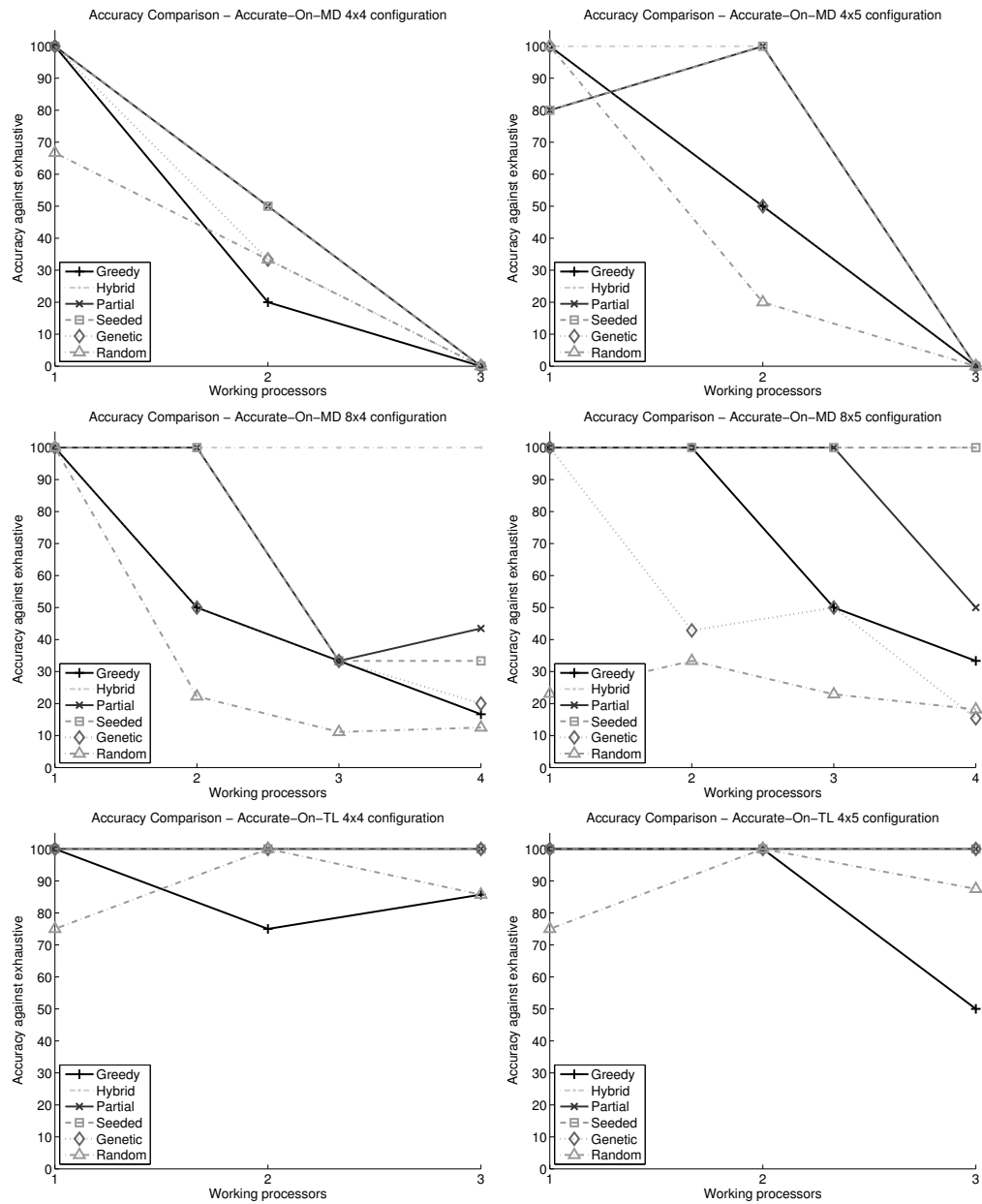


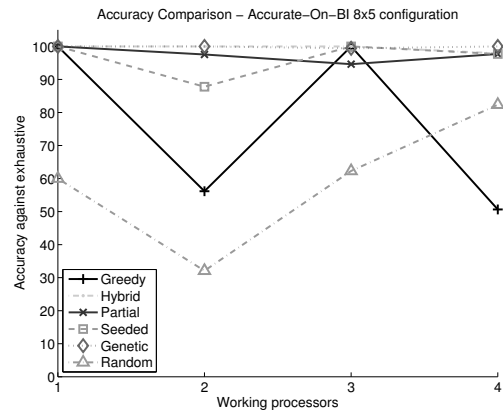
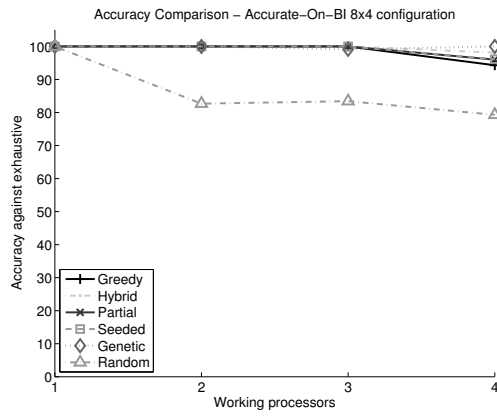
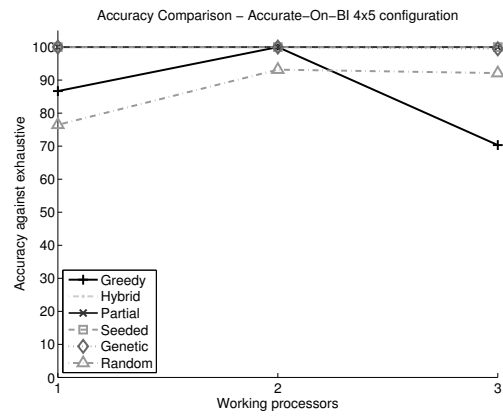
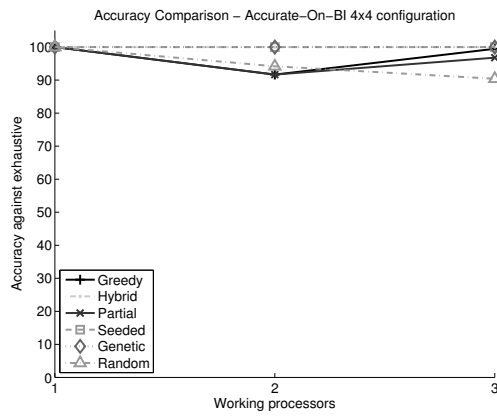
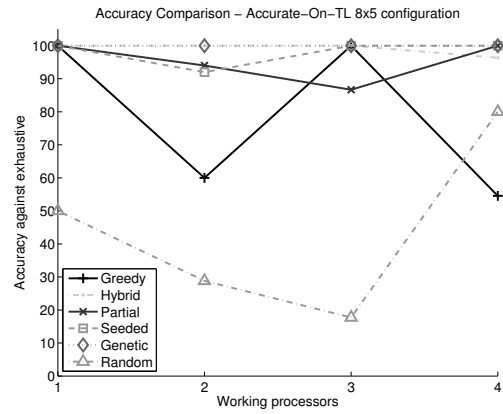
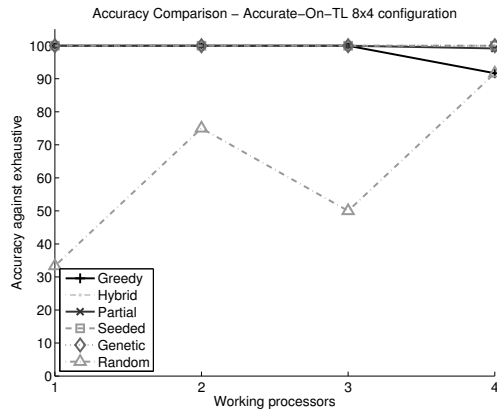


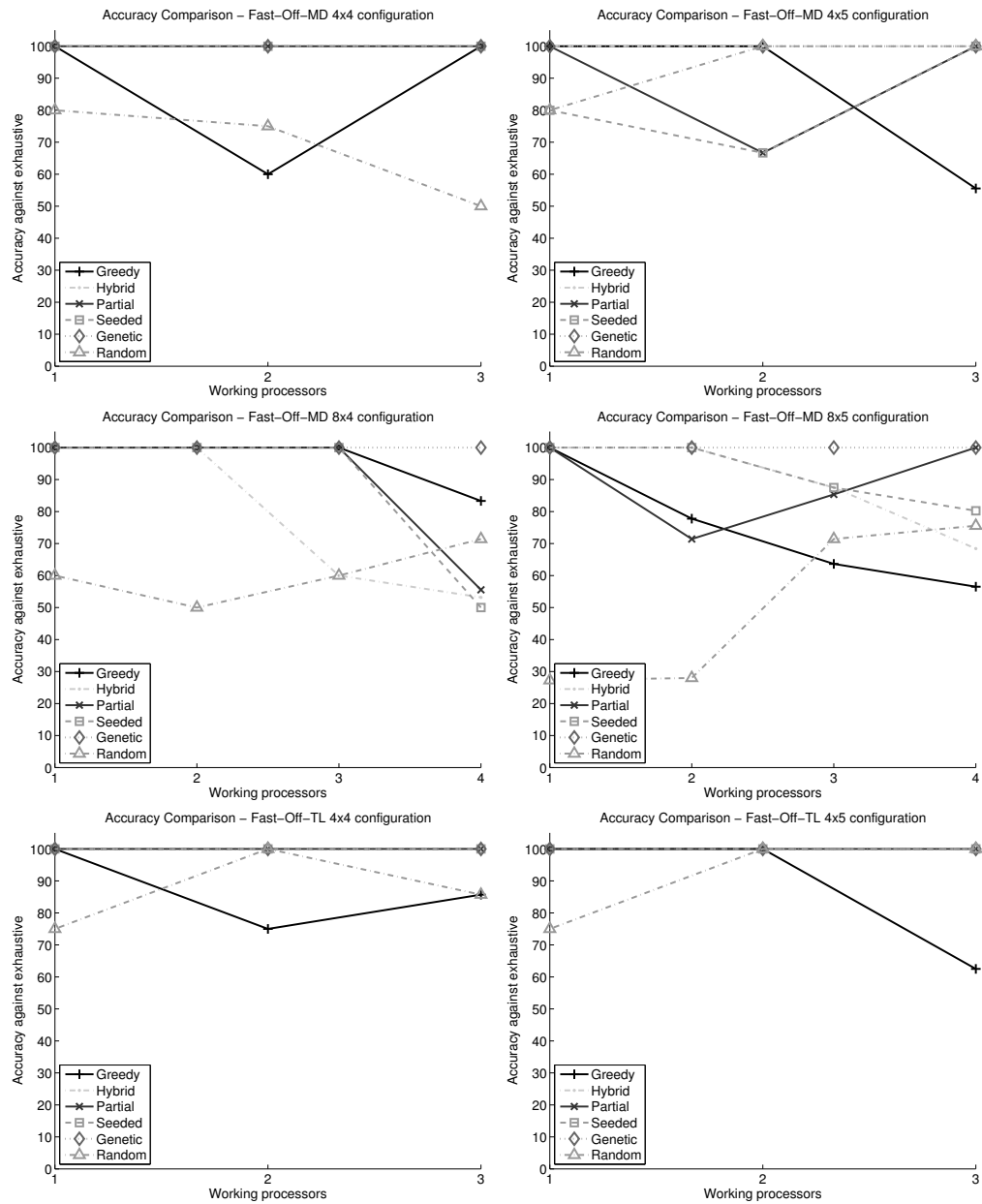
A.4 Solution accuracy plots

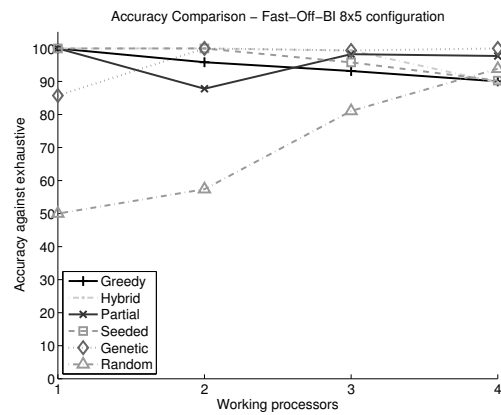
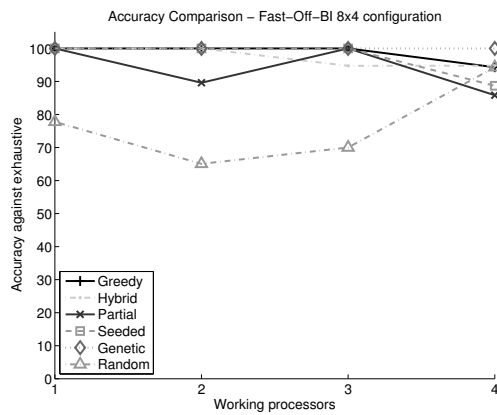
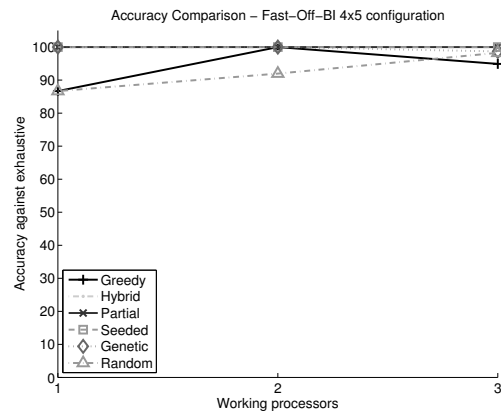
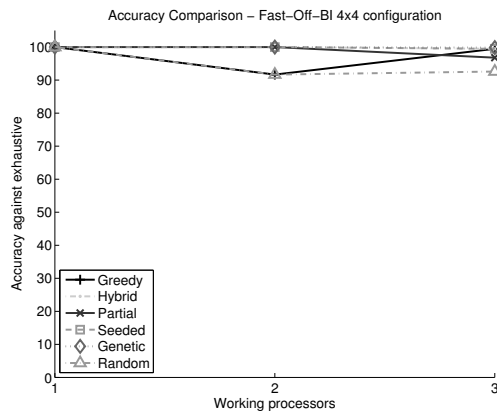
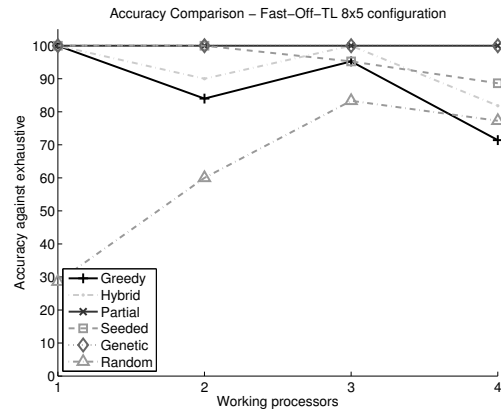
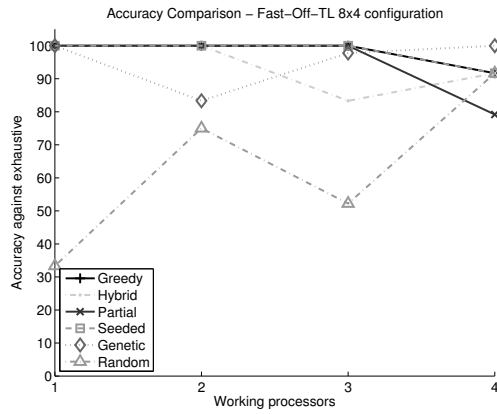


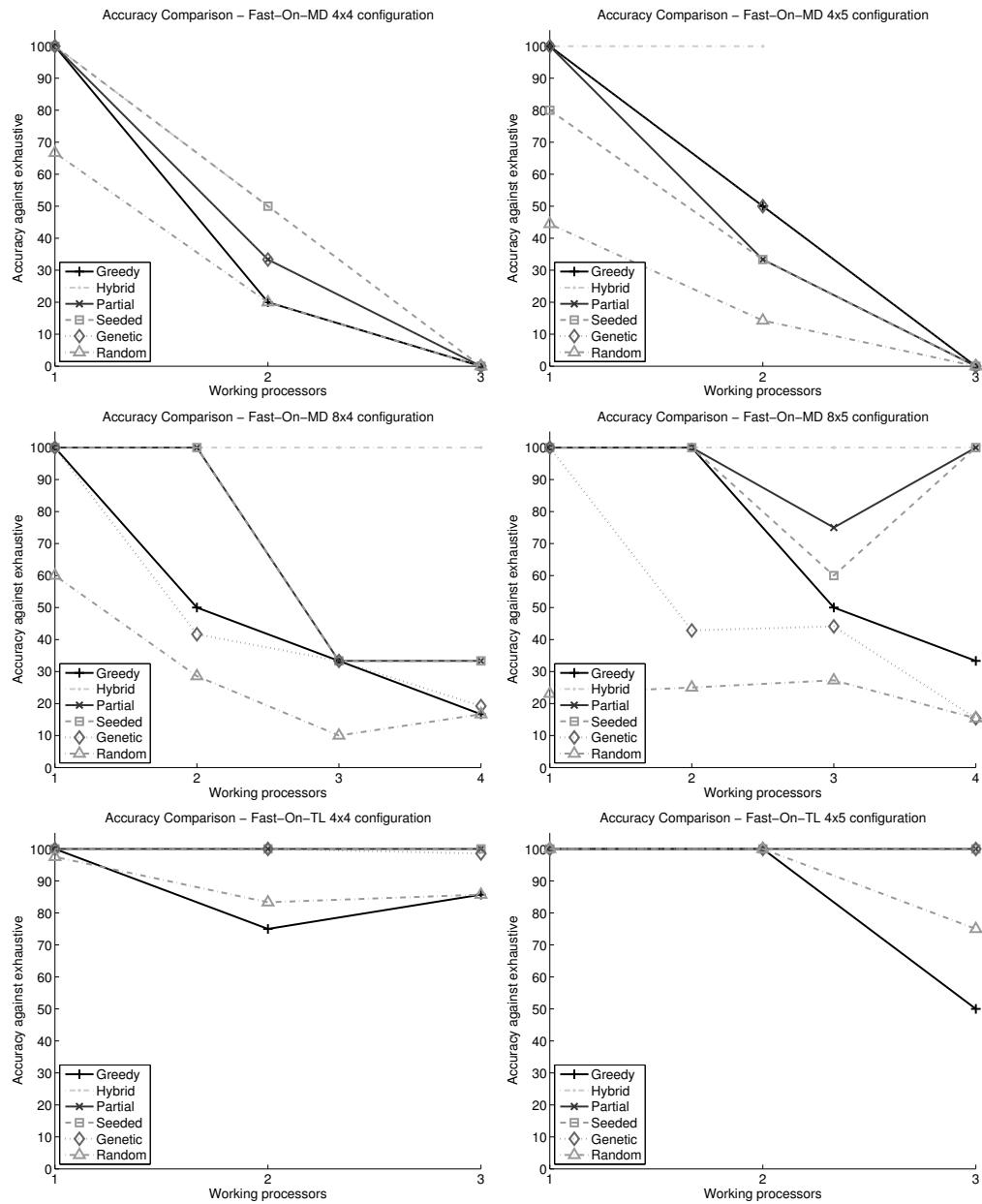


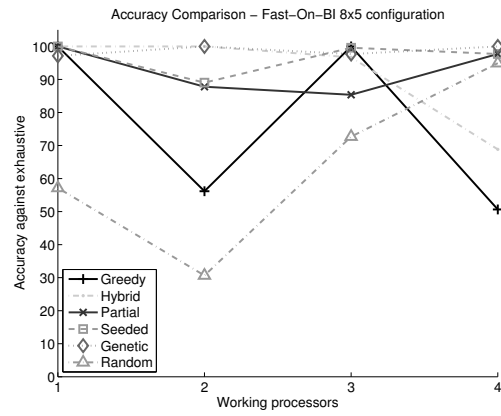
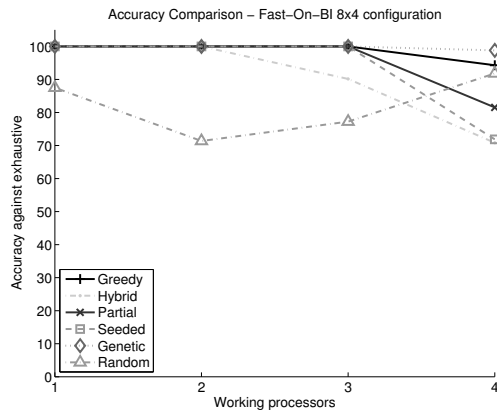
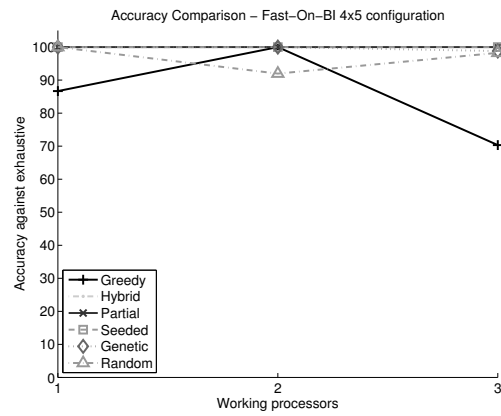
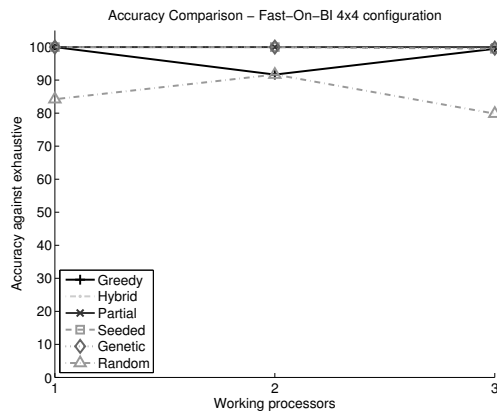
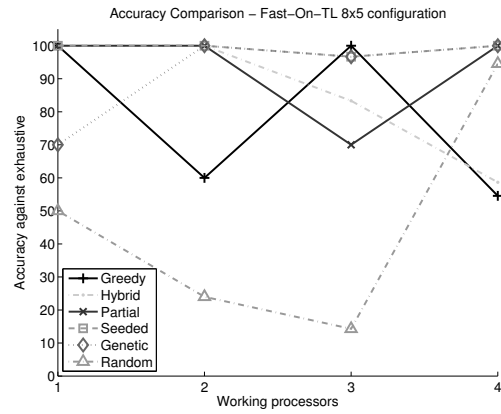
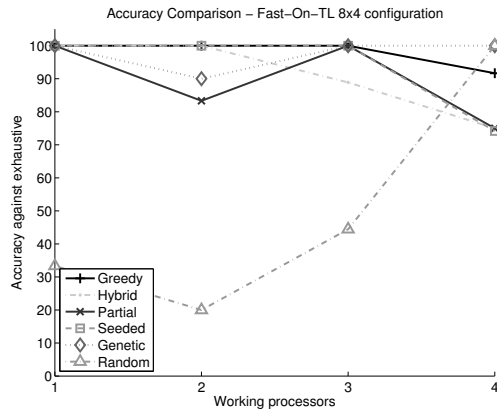












Curriculum Vitae



Vasileios Vasilikos was born in Thessaloniki, Greece, on December 13th 1985. He graduated from the Deutsche Schule Thessaloniki (DST) in 2003 with a GPA of 17.8/20.0 (Very Good).

After participating in the national examinations for a university education, he was admitted to the Technical University of Crete (TUC), in Chania, in the department of Electronics & Computer Engineering. He graduated with a GPA of 7.56/10.0 (Very Good) in July of 2009. The title of his Diploma Thesis was “Heuristic Search Optimization using Recursive Algorithm Selection and Reinforcement Learning” under the supervision of Michail G. Lagoudakis. The work of this thesis resulted in a journal publication in December 2010 in “Annals of Mathematics and Artificial Intelligence”, Springer Editions.

While working on his Diploma Thesis, in August 2008 he was admitted to the Delft University of Technology (TUD), in the Netherlands, in the MSc program of Computer Engineering where he continues his studies until now. His research interests include heuristic search methods and algorithms, computer architecture, artificial intelligence and any interesting combination thereof.