

SINCRONIZAÇÃO DE THREADS EM  
HARDWARE SIMD



TEO MILANEZ BRANDÃO

**SINCRONIZAÇÃO DE THREADS EM  
HARDWARE SIMD**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

Julho de 2013



TEO MILANEZ BRANDÃO

**THREAD SYNCHRONIZATION IN SIMD  
HARDWARE**

Dissertation presented to the Graduate Program in Computing Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computing Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

July 2013

© 2013, Teo Milanez Brandão.  
Todos os direitos reservados.

Milanez Brandão, Teo

B817t Thread Synchronization in SIMD hardware / Teo  
Milanez Brandão. — Belo Horizonte, 2013  
xxv, 93 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of  
Minas Gerais

Orientador: Fernando Magno Quintão Pereira

1. Computação — Teses. 2. Arquitetura de  
computador — Teses. I. Orientador. II. Título. (043)

CDU 519.6\*21



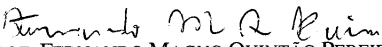
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO


Sincronização de threads em hardware SIMD


**TEO MILANEZ BRANDÃO**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. RENATO ANTÔNIO CELSO FERREIRA - Coorientador  
Departamento de Ciência da Computação - UFMG

  
PROF. SANDRO RIGO  
Instituto de Computação - UNICAMP

  
PROF. OMAR PARANAÍBA VILELA NETO  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 23 de agosto de 2013.





*Dedicuum cest laborae a quelquis personatum que ajudorat a facirelo.*



# Acknowledgments

I would like to thank my advisors, Professors Renato and Fernando. I thank Renato for receiving me in the course and I thank Fernando, because his knowledge and guidance were essential to the development of this work. He also helped me a lot to write paper and this dissertation.

Additionally, I am deeply grateful to Sylvain due to his ideas and knowledge that gave bases to this work.

Finally, I would like to show my gratitude to my colleagues because they helped me a lot during the time I was in the university.



*“Preventing is better than two ones flying.”*  
(Chapulín Colorado)



# Resumo

O desempenho é limitado pelo consumo de energia em arquiteturas de computadores modernas. Uma forma de reduzir o consumo de energia e aumentar o desempenho, é eliminar operações redundantes entre as instruções de máquina. Esta eliminação de redundância, contudo, é difícil, porque envolve a solução de um problema caro durante a execução: a super-sequência mais curta. Trabalhos anteriores propuseram muitas heurísticas diferentes para resolver este problema em nível de arquitetura ou em nível de compilador. O grande número de diferentes algoritmos, e o vasto espaço de busca fazem com que uma comparação entre eles uma tarefa hercúlea. Nesta dissertação, nós mergulhamos nesta tarefa, fornecendo a mais extensa análise comparativa destas diferentes heurísticas já vista na literatura. Nós combinamos as diferentes heurísticas em várias dimensões, incluindo a quantidade de paralelismo em nível de thread e em nível de dados. Os nossos resultados mostram que a heurística relativamente simples, tais como uma heurística chamada MinSP-MinPC, pode superar algoritmos muito complicados. A partir dessa comparação traçamos subsídios para projetar, testar e implementar novas heurísticas para compartilhar o trabalho redundante entre threads paralelas. Os nossos novos algoritmos melhoraram os trabalhos anteriores de maneiras não-triviais. Ao testar estes algoritmos em benchmarks de força industrial, nós observamos que alguns deles são capazes de reduzir o número de instruções a serem processadas por um fator de 3x.

**Palavras-chave:** Sincronização de Threads, Paralelismo em nível de threads, Paralelismo em nível de dados, SIMD, Compartilhamento de recursos de hardware, Redundância de instruções, Redundância de dados.





# Abstract

Performance is constrained by power consumption in modern computer architectures. A way to reduce power consumption and increase performance, is to eliminate redundant operations between assembly instructions. This redundancy elimination, however, is difficult, because it involves solving a costly on-line problem: the shortest common supersequence. Previous work have proposed many different heuristics to solve this problem at either the architecture, or at the compiler level. The sheer number of different algorithms, and the vast search space makes a comparison between them a herculean task. In this dissertation, we dive into this task, providing the most extensive comparative analysis of these different heuristics ever seen in the literature. We match the different heuristics along several dimensions, including the amount of thread-level or data-level parallelism that they deliver. Our results show that relatively simple heuristics, such as the so called MinSP-MinPC can outperform very convoluted algorithms. From this comparison we draw subsidies to design, test and implement new heuristics to share redundant work between parallel threads. Our new algorithms improve on the previous works in non-trivial ways. When testing these algorithms in industrial-strength benchmarks, we have observed that some of them are able to reduce the number of instructions to be processed by a factor of 3x.

**Palavras-chave:** Thread Synchronization, Thread-Level Parallelism, Data-Level Parallelism, SIMD, Hardware Resource Sharing, Instruction Redundancy, Data Redundancy.



# List of Figures

1.1	SIMD pipeline . . . . .	3
1.2	Example of different heuristics in control flow graph . . . . .	6
3.1	CFG of if-then-else with redundant code . . . . .	25
3.2	CFG of loop with continue statement and big code in the beginning . . . . .	25
3.3	CFG of loop that may end with break statement . . . . .	26
3.4	CFG of if without else block that calls a big function . . . . .	27
3.5	Example of a virtual function . . . . .	29
3.6	CFG with distant post-dominator . . . . .	33
3.7	CFG of loop with continue statement and big code in the end . . . . .	35
4.1	Example of trace . . . . .	46
5.1	Heuristics with blackscholes . . . . .	52
5.2	Heuristics with bodytrack . . . . .	53
5.3	Heuristics with fluidanimate . . . . .	54
5.4	Heuristics with swaptions . . . . .	55
5.5	Heuristics with tachyon . . . . .	56
5.6	Heuristics with barnes . . . . .	57
5.7	Heuristics with fft . . . . .	58
5.8	Heuristics with fmm . . . . .	59
5.9	Heuristics with ocean_ncp . . . . .	60
5.10	Heuristics with radix . . . . .	61
5.11	Heuristics with volrend . . . . .	62
5.12	Heuristics with water_nsquared . . . . .	63
5.13	Average result of heuristics . . . . .	64
5.14	Histogram (1/2) . . . . .	69
5.15	Histogram (2/2) . . . . .	70
5.16	Average of execute-identical instructions. . . . .	71

5.17	Memory access pattern (1/2)	72
5.18	Memory access pattern (2/2)	73
5.19	Average of memory access patterns	74
5.20	Memory access distances (1/4)	76
5.21	Memory access distances (2/4)	77
5.22	Memory access distances (3/4)	78
5.23	Memory access distances (4/4)	79

# List of Tables

4.1	Characteristics of the benchmarks. . . . .	45
5.1	Subtitles of benchmarks . . . . .	51
5.2	Subtitles of heuristics . . . . .	51
5.3	DLP of all benchmarks with all heuristics . . . . .	65
5.4	Throughout of all benchmarks with all heuristics . . . . .	66



# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Resumo</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The importance of Resource Sharing . . . . .	1
1.2 The MMT model . . . . .	2
1.3 Challenges of thread synchronization . . . . .	4
1.4 Heuristics . . . . .	5
1.5 Proposed contributions . . . . .	7
1.5.1 The New Heuristics . . . . .	7
1.5.2 The Extensive Comparison . . . . .	7
1.5.3 Measure of the amount of execute-identical instructions . . . . .	9
1.5.4 Analysis of memory access patterns . . . . .	9
1.6 The Rest of this Dissertation . . . . .	10
<b>2 Background and Related Works</b>	<b>11</b>
2.1 Some definitions . . . . .	11
2.2 Resource sharing at the hardware level . . . . .	12
2.3 A Brief History of SIMD hardware . . . . .	13
2.4 Divergence analysis . . . . .	14
2.5 Thread Reconvergence Heuristics . . . . .	15
2.6 Thread-level parallelism in SIMD . . . . .	18

2.7	Previous study of data redundancy in parallel applications . . . . .	20
2.8	Final considerations . . . . .	21
<b>3</b>	<b>The heuristics</b>	<b>23</b>
3.1	Hardware-level heuristics . . . . .	23
3.1.1	Minimum-PC . . . . .	24
3.1.2	Minimum SP, Minimum PC . . . . .	26
3.1.3	Maximum Function Level, Minimum PC . . . . .	28
3.1.4	Long . . . . .	29
3.1.5	Variations of Long's Heuristic . . . . .	31
3.1.6	Lee . . . . .	31
3.2	Heuristics that require ISA change or compiler support . . . . .	32
3.2.1	Nearest post-dominator reconvergence . . . . .	33
3.2.2	Heuristic with structured control flow graph . . . . .	36
3.2.3	Thread frontiers . . . . .	38
3.3	Experimental Heuristics . . . . .	40
3.3.1	Reconvergence Detection . . . . .	40
3.3.2	Distance . . . . .	41
3.3.3	Greedy Oracle . . . . .	41
3.3.4	Round-Robin . . . . .	41
<b>4</b>	<b>Methodology</b>	<b>43</b>
4.1	Benchmarks . . . . .	43
4.2	Execution . . . . .	45
4.3	Instrumentation with compiler support . . . . .	47
<b>5</b>	<b>Results</b>	<b>49</b>
5.1	Comparison Between the Effectiveness of the Heuristics . . . . .	50
5.1.1	Discussion . . . . .	67
5.2	Execute-identical instructions . . . . .	68
5.2.1	Discussion . . . . .	71
5.3	Memory access pattern . . . . .	71
5.3.1	Discussion . . . . .	74
5.4	Distance between memory accesses . . . . .	75
5.4.1	Discussion . . . . .	80
<b>6</b>	<b>Final Remarks</b>	<b>81</b>
6.1	Limitations of this work . . . . .	82



6.2	Future works . . . . .	82
6.3	Software . . . . .	85
	<b>Bibliography</b>	<b>87</b>



# Chapter 1

## Introduction

In this dissertation we will discuss new ways to improve resource sharing at the hardware level. With such purpose, in this section we introduce fundamental notions, such as resource sharing and minimal multi-threading. We then move on to explain the importance of keeping threads as much synchronized as possible in contemporary hardware architectures. And finally, we present the challenges and the proposed contributions.

Before we delve into the subtleties of resource sharing in parallel applications, we provide a bit of motivation behind our work. In current architectures, performance is constrained by power consumption. A way to reduce power consumption and increase performance is to eliminate redundant operations. In this work, we provide different ways to keep threads synchronized, with the purpose of maximizing the amount of instructions simultaneously common to all of them.

### 1.1 The importance of Resource Sharing

Resource sharing at the hardware level is an alternative that computer architects have been using to decrease the costs of highly parallel processors and the power consumption. As a testimony of this fact, we observe the rising popularity of GPGPU (General-Purpose computation on Graphics Processing Units), which is largely based on the SIMD execution model [48].

The Single Instruction Multiple Data (SIMD) execution model is a technique that allows the hardware to share resources. In this case, multiple processing units share the same instruction fetcher. The SIMD hardware has multiple processing units that perform the same operation on multiple data points simultaneously. By giving the same instruction to different processing units, the SIMD model fosters a programming

style strongly based on data parallelism. This sharing is essential to keep the energy consumption of GPUs low, while giving them performance capabilities never seen before in the hardware industry [38].

The SIMD architecture is not the only way to share resources among different threads. As another example of resource sharing, we have the superscalar architectures [28]. This type of hardware allows faster CPU throughput because it executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to the same redundant functional units of the processor. Examples of superscalar architectures are present in the x86 computers, which are used in servers and personal computers.

A third example of resource sharing at the hardware level is pipelining [28]. That technology is another way to increase the instruction throughput of the CPU. The execution is divided in stages and each stage has an execution unit. An instruction executes only one stage per cycle and the CPU will have multiple instructions executing a stage during a cycle. Pipelining is heavily used in the hardware industry and nowadays, virtually every processor uses some sort of pipeline to speedup the execution of programs.

Finally, we can also cite Simultaneous Multi-Threading as an example of beneficial resource sharing [65]. In this case, multiple independent threads can better utilize the resources provided by the processor. This improvement happens because a thread can use a resource while another thread is executing, as long as they do not use the same resource. The SMT (Simultaneous multi-threading) hardware is present in a number of architectures, for instance, the Hyper-threading technology, which was introduced in Intel Pentium 4.

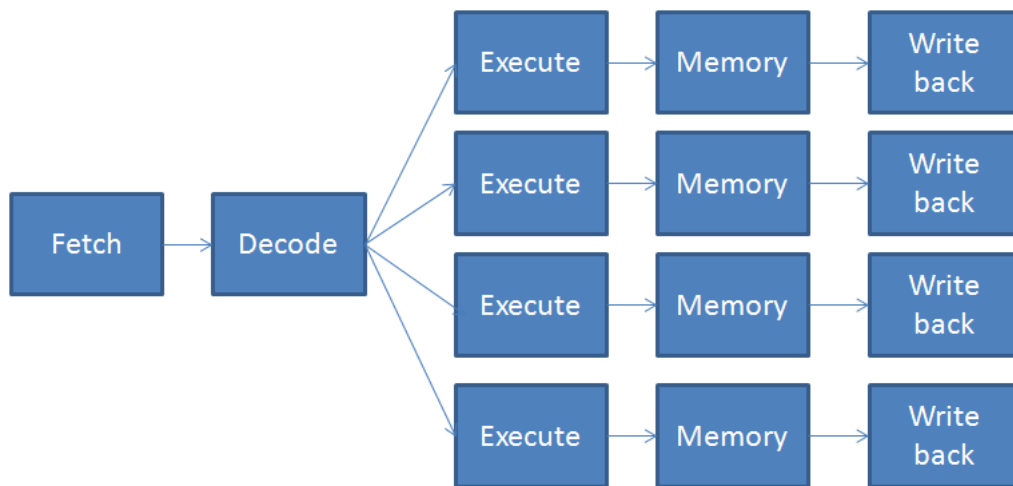
From these examples, it is clear a general tendency of the industry to invest time and resources in the design and development of sharing techniques at the hardware level. Given that most of the newer architectures have some way to share resources, and that new ways to portion out resources are being developed [39], we expect that this tendency will only continue to grow. Nevertheless, there are still challenges that must be overcome in order to maximize resource sharing at the hardware level.

## 1.2 The MMT model

In this dissertation we will pay particular attention to the Minimal Multi-Threading (MMT) execution model [39]. This is an important technique to share resources between threads. MMT is a mechanism that was recently proposed to combine

instructions of different threads when they are equal in a SMT machine. The MMT model executes threads with the same instruction simultaneously whenever it is possible.

An MMT-based architecture organizes threads into groups that share the same instruction fetch and decode unit, and might share execution units. Each thread keeps its own program counter (PC). At fetch time, the hardware chooses heuristically the next PC among all the PCs of active threads. The next instruction to be processed will be fetched at this PC. If the chosen PC is the same across several threads, then all of these threads receive an instruction to execute. If this instruction has the same input values, then the computations might be combined as well, so the instruction is issued once on behalf of all participating threads.



**Figure 1.1.** SIMD pipeline

Minimal Multi-threading, being a recent concept, still offers room for improvements. The SIMD model, which is illustrated in figure 1.1, executes different threads simultaneously only when all these threads share the same instruction. From this observation comes the need to keep the threads synchronized, so that they might process the same instruction as often as possible. We are interested in maximizing the amount of common instructions executed by independent threads. Keeping threads as much synchronized as possible is important because if two separate threads read different program counters, then they will compete for the shared pipeline front-end, in particular the fetch unit, causing pipeline stalls and/or increased energy consumption.

In its original conception, MMT does not explore any form of redundancies in memory access patterns. The reason for this limitation is simply the fact that researchers have not yet demonstrated that such redundancies are common in SIMD

programs. Nevertheless, this type of redundancy has been already acknowledged in the GPU world as a promising way to save hardware space and to reduce energy consumption [39].

Redundancy can be found in instructions and data. Instructions redundancy happens when multiple threads execute the same instruction and data redundancy happens when these redundant instructions use the same data as input, which can be memory address or the same register value. A study done by Long [39] seeks to find redundancies in the instructions of different threads. Long classifies the redundant instructions as follows:

- The instructions are the same (**fetch-identical**): The PC of these instructions is the same, therefore the instruction have to be fetched from the memory only once. The SIMD machine can only run simultaneously the threads that are fetch-identical.
- The instructions have the same result (**execute-identical**): Besides being fetch-identical, these instructions have the same input data, in other words, the data in each register that are read by the instruction is the same in all threads and consequently the result will be the same. It is mostly discovered during the execution of the program, but there are cases that can be detected at compile-time, such as instructions that manages *for* loops.

### 1.3 Challenges of thread synchronization

It is difficult to find an universal synchronization heuristic that is good for every type of application, because each program has its specific characteristics. Furthermore, the programs may be written in any programming language, including assembly, and may be strongly optimized by compilers. The resulting program may use some instructions for unusual purposes and may control the function stack by itself. As a consequence, it may have unstructured control flow. Given this diversity, we have observed that it is often the case that a thread synchronization heuristic that works for a program yields bad results for a different one.

Another problem is that the hardware implementation of a heuristic may be very expensive due to the complexity of physical components or due to the performance in hardware. In our work, we will simulate the heuristic in software and count the number of fetched instructions that are shared to measure the performance. However when a heuristic is implemented in hardware, it may take a long time to choose the PC to fetch instructions, despite having good results in the software simulation. In

other words, we will be approximating the behavior of a hardware, using a rather contrived metric: the number of fetched instructions. A deeper evaluation of these heuristics would have to take into consideration also the complexity of implementing these heuristics in hardware. This omission is a limitation of our study.

As an example of the impact of a heuristics on the performance of the hardware, we can analyse Long's reconvergence algorithm. Long's original formulation [39] uses an intricate reconvergence heuristic, which, in the words of the authors themselves, has impact on the performance of the hardware. This heuristics is expensive because it looks up the history of program counters on every execution cycle. On the other hand, the solution that has been currently used for GPGPU has very limiting constraints when we want to extend the SIMD execution model to general-purpose processors [11].

Function-level synchronization poses more problems to thread reconvergence, because the program may have recursive function calls and indirect function calls. We call a function invocation indirect if that function is activated through a pointer which might not be known at compilation time. Two threads may execute the same indirect call instruction and call different functions. These functions may call a function *foo* that is the same in both threads. In this case, it would be difficult to synchronize the execution of the function *foo* that both threads execute. Indirect calls are common in object-oriented programs. In this case, the shared function belongs to a base class and may be used by objects of different subclasses.

## 1.4 Heuristics

The synchronization problem may be initially seen as the shortest common supersequence, whose optimal solution is NP-hard [4]. This problem is defined as: *"given two or more sequences of symbols, find the smallest supersequence that includes all those sequences. The solution must keep the relative order of the symbols in each sequence, although the symbols may be contiguous or not"*. In this context, the sequences are the threads and the symbols are the PCs of the instructions.

To exemplify this problem, we have the following sequences of symbols:

1. ABCDE
2. FBCAG
3. EFDAG

The problem is solved in the following way:

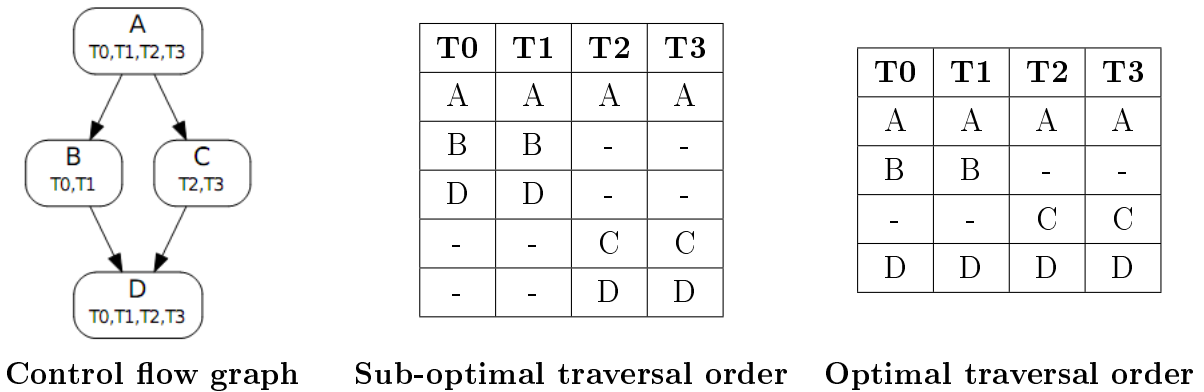
```
A--BCDE--
--FBC--AG
-EF--D-AG
```

And the result, the shortest common supersequence, is:

```
AEFBCDEAG
```

This solution yields the smallest combined sequence. Additionally, every input sequence is included in the solution. That is, the relative order of the symbols of each input sequence is preserved in the final solution of this problem.

Our problem, however, has one extra difficulty: we do not have all symbols during the execution of the programs. In other words, we are dealing with an on-line problem. We cannot foresee the instructions that will be executed, and therefore the best existing heuristics that finds a reasonable result for the shortest supersequence problem cannot be used in hardware. Thus, we will need to use a good heuristic (or policy) to schedule common threads according to the current instructions without needing to know the future instructions. In figure 1.2, we can see the importance of a good heuristic.



**Figure 1.2.** Example of different heuristics in control flow graph

We will be crafting different heuristics that are classified in two distinct groups. The first group contains the heuristics that are implemented without changes in ISA (Instruction-Set Architecture) and are compiler-independent. The other group is made of heuristics that propose changes in ISA or compiler. Details about these groups and the heuristics are explained in chapter 3.



## 1.5 Proposed contributions

The goal of this dissertation is to propose, implement and evaluate different thread synchronization heuristics. We hope, with this study, to advance the research related to the MMT execution model. To meet this goal, we will explore different ways to share resources in parallel applications. Below we list the four main contributions that come out of this work:

- New thread synchronization heuristics, which are competitive with the state-of-the-art algorithms already described in the literature.
- The most extensive comparison between different thread synchronization heuristics in the literature, in terms of data-level parallelism and thread-level parallelism.
- Measure of the amount of execute-identical instructions.
- Analysis of memory access patterns.

### 1.5.1 The New Heuristics

During this research we have designed and implemented a number of different thread synchronization heuristics. The participation of Sylvain Collange, currently a researcher at INRIA in Rennes, was paramount in this effort. Each of our new heuristics will be explained in chapter 3. They are:

- MinSP-MinPC [11, 44] and other heuristics based on this idea.
- Greedy Oracle
- Distance

### 1.5.2 The Extensive Comparison

Perhaps the most important contribution of this work was the very extensive comparison of different thread synchronization heuristics. We have compared these heuristics assuming different execution models. This work tries to study a hybrid of SIMD and MIMD architectures [7]. The design models we have used in our study are listed below:

- **Pure MIMD**: all threads advance independently as fast as possible. It implies maximal parallelism, but no instruction sharing.

- **Pure SIMD:** always synchronize threads to share instructions. It implies no instruction parallelism, but maximal instruction sharing.
- **Opportunistic SIMD:** same as MIMD, but shares instructions when possible. It implies high instruction parallelism and high instruction sharing. It is a theoretic model, but it can be implemented as a MIMD machine that is able to share fetch and decode units while other units are unused.

Brunie [7] has shown performance improvement going from SIMD to "2IMD", which is fetching 2 different instructions per cycle. Now, we want to know the benefits of going to 3IMD, 4IMD, until MIMD. Thus, we will study the opportunistic SIMD model. Each heuristic gives a different tradeoff between the ideal opportunistic SIMD and the ideal pure SIMD. This tradeoff is measured by three metrics:

- **Data-level parallelism (DLP)** is the average number of fetch-identical instructions (or the number of active threads) over the number of cycles. It is only possible when we have threads with same instruction, but the data can be different. It implies instruction sharing, which indicates energy efficiency. In an SIMD hardware, the highest DLP implies the best performance.
- **Thread-level parallelism (TLP)** happens when we have multiple threads with different instructions. The different instructions are only executed in parallel in MIMD hardware while they are serialized in SIMD hardware.
- **Instruction throughput** indicates performance in opportunistic SIMD architecture. It equals the average number of executing threads per cycle and it equals DLP times TLP.

DLP implies the total number of instructions fetched during the execution of an application. The larger is the DLP, the smaller is this number. In other words, it equals the total number of instructions executed by all threads divided by the number of fetched instructions. We want the highest possible DLP to achieve the best performance in the SIMD hardware. The highest DLP also gives us the highest level of instruction sharing, which are fetch-identical instructions and they may be execute-identical instructions. If all instructions had the same execution time, the DLP would be the speedup in SIMD in comparison to serial execution.

We also want to find the tradeoff between DLP and TLP that optimizes the DLP and the instruction throughput for the opportunistic SIMD hardware. Thus, we are pursuing a setup that gives us good instruction sharing and good performance. From

the instruction sharing we get low energy consumption. Performance, needless to say, is the holy grail of the computer architect.

### 1.5.3 Measure of the amount of execute-identical instructions

The analysis of execute-identical instructions allows us to identify data redundancy that may help to propose ways to optimize the hardware. This optimization may reduce the cost or save energy.

The common cases of execute-identical instructions refer to opcodes that do not read register or do not have parameters. It includes unconditional branches, for instance. In the other cases, the registers that are read have the same data in all threads and consequently the result of the instruction is the same. As an example, we can include read from the same memory address and the counter of iterations in loops like *for*. Direct function calls and return instructions may be treated as execute-identical despite the value of SP (Stack pointer) and FP (frame pointer) being different, but they are the same in relation to the base of the stack.

### 1.5.4 Analysis of memory access patterns

Another direction of our research involves a study of the redundancy of data among different threads. In this dissertation we have focused on the MMT execution model. Thus, we analyse the amount of redundancy between data processed by different threads in this hardware. Our analyses uses a suite of typical software applications, which we believe is comprehensive enough to support our findings. The product of this study is an analysis of the memory access patterns [12] in the MMT setting. Such patterns describe the relative arrangement of addresses in the load and store instructions used by each thread. When the same instruction is executed by different threads simultaneously and it accesses the memory, the addresses that are accessed by different threads may have some patterns. We classify these patterns in the following way:

- **Uniform:** All threads access the same memory address. Of course, they are read operation.
- **Affine:** The threads access different addresses, but the addresses are equally spaced in memory. The address accessed by a thread is:  $\text{Address} = \text{base} + \text{delta} * \text{tid}$ , where base and delta are constant values. We also include here the cases where all threads accesses the same relative address in its own stack. This pattern is classified according to the size of the affine distance of the memory accesses.

- **Scattered:** The threads access addresses that are different and scattered. This pattern is classified according to the size of the interval where all memory accesses are done.

As we show in this dissertation, we have observed substantial regularity in inter-thread access patterns. This fact motivates the adoption, in the MMT context, of recent memory coalescing hardware mechanisms that have been proposed for GPUs [14]. For instance, if all the threads read data from the same location, or from regularly spaced locations, then the hardware can bring all this data to registers with only one cache access. On the other hand, if simultaneous memory accesses are randomly scattered, then we lose inter-thread locality, and access fragmentation puts an increased pressure on caches.

Current multi-threaded hardware has not been designed to benefit from uniform and affine memory patterns: independent on the target address,  $n$  simultaneous threads require  $n$  accesses to memory ports. However, there are proposals for new hardware designs that proceed differently [14]. In these processors, a uniform address causes only one access to the data cache. It is this kind of hardware that will mostly benefit from this study of memory access patterns.

Benefiting from regularly-spaced accesses is more complicated, but it is not impossible. If  $n$  threads execute an affine access, then a set of  $n$  memory cells, spaced by a constant distance  $K$ , and starting at base address  $C$  is accessed at once. When  $K$  is equal to the word size, accesses are contiguous and can be combined into a single memory transaction. For other  $K$  values, it is also possible to have conflict-free parallel access to a banked cache [59].

The size of scattered accesses interval is also important. If the data simultaneously accessed by active threads is within a short distance of each other, then it may fit into the same cache line. In this case, if the data is already cached, then every thread scores a hit. Otherwise, it can be brought to the cache with just one trip to a lower level in the memory hierarchy.

## 1.6 The Rest of this Dissertation

The remainder of this dissertation is organized as follows. We will first present background and discuss related works in Chapter 2. In Chapter 3, we describe all the heuristics that we have used in this research. Chapter 4 explains the methodology that we use in our experiments. In Chapter 5, we show the results of the simulation and discuss them. Finally, Chapter 6 concludes our work.

# Chapter 2

## Background and Related Works

The literature contains many different examples of research whose goal is to explore resource sharing to improve speed or energy consumption in parallel architecture. In this dissertation we review some of these works. Before we delve into them, we will define a few terms that will be used throughout this dissertation.

### 2.1 Some definitions

According to Alle [2], a basic block is a portion of the code within a program with certain desirable properties that make it highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. The code in a basic block has one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program, and one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block. Under these circumstances, whenever a basic block starts, all instructions of the basic block are necessarily executed.

Basic blocks form the vertices or nodes in a control flow graph. A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves [2].

A thread in the context of this dissertation is a sequence of programmed instructions that can be executed independently in the computer. Multiple threads

can exist within the same process and share resources such as memory, while different processes do not share these resources. In a SIMD program, we have multiple threads executing the same code simultaneously, but with different inputs. The different threads can take different paths in the same control flow graph of the code due to different data.

## 2.2 Resource sharing at the hardware level

Instruction and data sharing are not new ideas in the computer architecture world. In the mid nineties, Tullsen introduced the notion of Simultaneous Multi-Threading (SMT) [65]. In the SMT execution model, several threads share the same superscalar pipeline, including the front-end fetching and decoding instructions. In this way, the hardware is better equipped to avoid control and data hazards; hence, keeping the many stages of its pipeline always in use.

Intra-thread redundancy exists if different instructions within the same thread of execution share the same elements, e.g., registers, data or addresses. Intra-thread instruction redundancy has been exploited by memorizing values for later reuse. It includes the reuse of results of long-latency logic-arithmetic operations [54, 55, 9] and the compression of functions or arbitrary sequences of instructions into a single memorized result [24, 68, 15, 52, 67, 23]. Further studies were performed to analyse the effectiveness and propose solutions for instruction reuse in the hardware [60, 61, 45, 10, 63, 3, 50, 46]. These optimizations increase performance and reduce energy consumption [39].

In this work we will be concerned about inter-thread redundancy, which is commonality that exists across different threads of execution. Inter-thread instruction redundancy has also been exploited in previous works. Inter-thread techniques were used to reduce cache misses [8, 27, 6]. Acar [1] proposed a software approach by maintaining the dependence graph of the computation and using the graph to propagate changes from inputs to the output. Dechene [18] studied multiprocessors to evaluate a hardware implementation that recognizes and exploits instruction-similarity.

In 2008, Gonzalez [25] brought in the concept of Thread Fusion, as a way to decrease the energy consumption of SMT machines. Thread fusion consists in giving the same instruction to different threads, whenever they have the same program counter; hence, providing a way to share the pipeline front-end. In order to reconverge threads, Gonzalez would require the compiler to insert barriers at control independent program points.

Long [39] extended Gonzalez's work by introducing the idea of minimal multi-threading. They have designed a hardware mechanism that reconverges threads without the intervention of a compiler. Notice that whereas SMT implies resource multiplexing, MMT strives for resource sharing. That is, in the SMT case, only one thread can use a given resource at a given time. In the latter, several threads cooperatively use a resource to perform the same action, promising higher energy reductions than what could be achieved with independent thread execution.

The literature describes a few different ways to share the same hardware resource between different processing cores. The goal of this sharing is the reduction of energy consumption and the reduction of the manufacturing cost. Kumar [33] studied the advantages and disadvantages of sharing FPU (floating point unit), instruction cache and data cache between different cores. The sharing of instruction cache is good when the threads execute the same program code. The sharing of FPU is advantageous in programs where floating point operations are not frequent. In this case, it is likely that both threads will not need the floating point unit at the same time. Therefore, they can re-use it between themselves. If a conflict happens, then one thread must wait for the other.

## 2.3 A Brief History of SIMD hardware

GPGPU applications brought renewed interest to the SIMD execution model. Researchers have developed a plethora of applications for GPUs, and this effort required a substantial amount of a pattern of behavior that we call "SIMD thinking". As we will discuss shortly, the development of SIMD applications is substantially different from the development of applications in other execution models. These differences and peculiarities constitute what we call SIMD thinking. Although today we see many discussions about this model, its history is much older.

The first use of SIMD instructions was in vector supercomputers of the early 1970s such as the 'CDC Star-100' and the 'Texas Instruments ASC', which were able to operate on a "vector" of data with a single instruction. Vector processing was made popular by Cray in the 1970s and 1980s. Vector-processing architectures are now considered separate from SIMD machines, based on the fact that vector machines processed the vectors one word at a time through pipelined processors (though still based on a single instruction), whereas modern SIMD machines process all elements of the vector simultaneously [51].

Early SIMD machines were characterized by massively parallel processing-style

supercomputers such as the 'Thinking Machines' 'CM-1' and 'CM-2'. These machines had many limited processors that would work in parallel. For example, each of the 64,000 processors in a Thinking Machines CM-2 would execute the same instruction at the same time, allowing, for instance multiplications on 64,000 pairs of numbers at a time.

The first SIMD machines were unable to run programs with loops and decision structures. There were machines whose threads executed instructions that were chosen by a master thread, which has loops and decision structures. These machines were limited because the threads cannot diverge. Later, some machines able to put up with divergence appeared. With this new capacity came the challenge to reconverge divergent threads. This challenge has motivated a long string of research in thread reconvergence heuristics.

## 2.4 Divergence analysis

Since SIMD hardware is used in GPU nowadays, many researchers want to optimize its performance and remove its limitations. Divergence is the source of performance degradation in SIMD hardware, because the threads follow different paths after executing a branch instruction and the hardware issues only one instruction at a time. Thus, some threads will have to wait, idly, while other threads execute. Divergence cannot be completely remove, but it can be reduced. Therefore, the study of the divergences is important to find ways to improve the performance.

To do analysis and characterization of applications to assist future work in compiler optimizations, application re-structuring, and micro-architecture design. Kerr [31] proposed a set of metrics for GPU workloads and used these metrics to analyze the behavior of GPU programs. The results quantify the importance of optimizations such as those for branch reconvergence, the prevalence of sharing between threads, and highlights opportunities for additional parallelism.

According to Coutinho [17], optimizing the code to avoid divergences is difficult because this task demands a deep understanding of programs that might be large and convoluted. To facilitate the detection of divergences, Coutinho introduces the divergence map, a data structure that indicates the location and the volume of divergences in a program. This map is built via dynamic profiling techniques. To illustrate the importance of the divergence map, he used it to pinpoint the core regions that must be optimized in well-known public applications. By hand optimizing some applications, he added 9–11% speedups onto kernels that have already gone through



the sieve of many programmers.

In face of divergences, Coutinho [16] introduced divergence analysis, a static analysis that determines which program variables will have the same values for every thread. This analysis is useful in three different ways: it improves the translation of SIMD code to non-SIMD CPUs, it helps developers to manually improve their SIMD applications, and it also guides the compiler in the optimization of SIMD programs. He demonstrated this last point by introducing branch fusion, a new compiler optimization that identifies, via a gene sequencing algorithm, chains of similarities between divergent program paths, and weaves these paths together as much as possible. Branch fusion is a technique to remove redundant code to decrease the size of divergent paths.

Many of these optimizations rely on divergence analysis, which classify variables as uniform, if they have the same value on every thread, or divergent, if they might not. Sampaio [57] introduced a new kind of divergence analysis, that is able to represent variables as affine functions of thread identifiers. The experiments show that this algorithm reports 4% less divergent variables than the previous state-of-the-art algorithm. Furthermore, we can mark about one fourth of all divergent variables as affine functions of thread identifiers. He also introduced the notion of a divergence aware register allocator. This allocator uses information from this analysis to either rematerialize affine variables, or to move uniform variables to shared memory. As a testimony of its effectiveness, this divergence aware allocator produces GPU code that is 29.70% faster than the code produced by Ocelot’s register allocator.

Another work to reduce branch divergence was done by Han [26], which proposed two software-based optimizations: iteration delaying and branch distribution. Iteration delaying targets a divergent branch enclosed by a loop within a kernel. It improves performance by executing loop iterations that take the same branch direction and delaying those that take the other direction until later iterations. Branch distribution reduces the length of divergent code by factoring out structurally similar code from the branch paths. The evaluation of the two optimizations shows that they improve the performance of the synthetic benchmarks by as much as 30% and 80% respectively, and that of the real-world application by 12% and 16% respectively.

## 2.5 Thread Reconvergence Heuristics

The main goal of our work is to design new thread reconvergence heuristics, and to compare these new approaches against older methods. Many different reconvergence algorithms have been described in the literature, and in this section we present a few

of them. Some of the methods that we discuss here have been designed for the early SIMD machines. Others are more recent, and have been developed to modern Graphics Processing Units.

In 1984, Lorie-Strong [40] from IBM proposed what we believe is the first thread reconvergence heuristic ever described in the literature. In the words of Collange [11], in Lorie-Strong's solution, the compiler performs a relaxed topological sort on the control flow graph and labels all basic blocks following this order. This numbering controls both the order of traversal of the control flow graph and the detection of reconvergence. The processor maintains both a PC and a block number per thread. In the case of divergence, the block with the smallest number will be run in priority. When a potential reconvergence point is reached, the block number of each thread is compared with the number of the next block the processor is about to execute. When a match is found, the associated thread is re-enabled. The divergence control mechanism is exposed at the architectural level. The order of traversal of the control flow graph is set statically during compile time. The integrated GPU of Intel's Sandy Bridge processor [30] uses an approach that resembles the Lorie-Strong's method. These methods do not handle recursive functions and indirect calls.

Another technique proposed by Takahashi in 1997 allows running code in SIMD mode without requiring annotations in the instruction set [64]. In the words of Collange [11], each thread has its own PC. A control unit walks through the control flow graph. When a branch is encountered, priority is given to the branch whose entry point has the lowest address, as long as at least one thread is active. When no thread is active, the other branch is taken. The assumption that appears to be made is that reconvergence points are found at the "lowest" point of the code they dominate, that is at the highest address. The strategy consists in attempting to always execute the instructions whose addresses are lowest when deciding which branch to execute, in order to not pass a potential reconvergence point. The drawback of this method is that thread activity signals are delayed by one clock cycle as they are sent back to the control unit. At the time the control unit receives the value, the data associated with the last branch instruction are not available any more. The processor has to continue running the program while all threads are inactive. Else blocks are always run regardless of divergence, and loops always run one extra iteration. Besides, the fact that the processor can speculatively follow branches which are not taken by any thread leaves the possibility that it encounters invalid instructions or runs past the code section. Unlike the case of branch predictors in superscalar processors, no provision is made to return to a non-speculative former state. To circumvent those problems, the authors propose that branch instructions be duplicated by the compiler. However, this

cancels the advantage of not depending on the compiler.

In 1988, Quinn [53] proposed a heuristic that chooses the thread with the smallest PC value to execute. The processor then just needs to compare the PC of each thread against the global PC to find active threads. Reconvergence happens when the PCs of different threads coincide. Further details are presented in section 3.1.1.

Some reconvergence techniques require the program to have a structured control flow graph (CFG). A program with structured CFG only contains sequence of instructions, conditional blocks (such as if-then and if-then-else) and loops (such as while and do-while). These loops can be nested. An unstructured CFG contains arbitrary jumps to any part of the program and they appear in programs written with structured programming languages due to compiler optimizations. It is possible to transform an arbitrary CFG into a structured CFG at compile time, at the possible expense of replicated code when the graph is not reducible [69]. The transformation method was created by Zhang [71] in 2004.

Levinthal [37] proposed the Pixar Chap Computer in 1984, which synchronizes programs with structured control flow with if-then-else and do-while statements. It supports nested structures using stacks where the push and pop operations are provided by a change in the ISA. Later, Chap project was improved in the POMP parallel computer [32]. It replaces the stack by counters that holds the current depth level of each thread. The active threads have the highest depth level. Unlike Chap, POMP supports recursive function calls, but the stack pointer of the active threads must be the same.

Intel G45 [29] improved the ideas of POMP including the use of explicit instructions to represent *continue* and *break* statements to allow to exit from loops with any nesting depth. The structured control flow graph is more flexible and allows smaller program codes. Indirect calls are not supported, but they are not a challenge for next versions. A heuristic that uses structured control flow graphs with break and continue is presented in section 3.2.2.

More recently, NVIDIA has developed the Tesla GPU architecture [38]. In the words of Collange [11], Tesla's instruction set includes conditional branch instructions resembling those of scalar processors, rather than structured control instructions. These instructions are complemented by annotations pointing at divergence and reconvergence points. Divergence is handled using a stack-based scheme, as in Chap. However, Tesla provides less information in program code than other instruction set. In particular, the matching between divergence points and reconvergence points is not explicit in the binary. Hence, the divergence stack needs to store the address of reconvergence points in addition to masks. This excludes an implementation based on

activity counters. Later, the mechanism used by Tesla can be extended to support indirect jumps, as offered by the Fermi architecture [48]. Additionally, this approach gives some flexibility in the traversal order that other techniques do not allow. It tends to move part of the scheduling decisions from the architecture to the micro-architecture. Coon [14] introduced explicit instructions to represent continue and break statements to help in the reconvergence.

Fung [22] proposed synchronization in the immediate post-dominator of a branch in 2007. A divergence instruction has multiple basic blocks as successor and the one to be executed is chosen by a conditional instruction. The immediate post-dominator is the nearest basic block that will be always executed after the divergence instruction with any path that the control flow takes. The Fung’s method supports unstructured control flow, but it has excessive thread serialization. Later, Fung [21] developed many methods to improve his heuristic, such as thread block compaction, but there is no generic method to improve all unstructured control flow cases. Our implementation of the idea of post-dominator is presented in section 3.2.1.

Brunie [7] also proposed a method to synchronize threads in the immediate post-dominator. The implementation is cheap, but the method has limitations. The only entry-point of the code block between the divergent instruction (PCdiv) and the post-dominator (PCrec) is the first branch instruction. In this technique, each PCdiv contains a pointer to PCrec and while any thread is between PCdiv and PCrec, the other threads will be blocked in PCrec. Furthermore, PCdiv must be smaller than PCrec. Thus, this technique does not allow synchronization of loops.

Diamos [19] proposed thread frontiers in which the compiler gives priority to the basic blocks and the hardware chooses the threads where the basic block has the highest priority. This is currently the state-of-the-art in thread synchronization for GPGPU. This method is cheaper to implement and faster to choose threads to execute than other simple heuristics, such as Quinn’s heuristic. This heuristic does not present a specific way to handle function calls, but it can use other methods to help. Most synchronization methods for GPGPU use a mechanism based on stack or counters, which Chap or POMP use to handle divergences when some threads call a function while other threads do not. Further details about thread frontiers are presented in section 3.2.3.

## 2.6 Thread-level parallelism in SIMD

In this work, we explore different ways to increase a metric called Thread Level Parallelism. This is the amount of different work that is done by different threads. A high TLP is present in non-regular programs and does not help SIMD machines. The measure of TLP with an optimized heuristic helps to identify the irregularity of SIMD programs. Previous work have provided ways to improve TLP, and in this section we go over some of those results.

We start our review with MLEP [49], multithreaded lockstep execution processor. This processor architecture improves the resource cost of SMT machines and the limited applicability of SIMD machines. This architecture exploits thread-level parallelism (TLP) as in SMT, but it executes parallel threads in lockstep as in SIMD by translating TLP into statically scheduled instruction level parallelism (ILP) by a compiler.

Other ideas to take advantage of higher TLP, besides MLEP, have been proposed. High TLP is not necessarily good in SIMD hardware, because the thread divergences cause under-utilization of the hardware potential. According to Narasiman [47], the GPU cores are still under-utilized, resulting in performance far short of what could be delivered. Two reasons for this are conditional branch instructions and stalls due to long latency operations. To improve GPU performance, computational resources must be more effectively utilized. To accomplish this, Narasiman [47] proposed two independent ideas: the large warp microarchitecture and two-level warp scheduling. Their mechanisms improve performance by 19.1% over traditional GPU cores for a wide variety of general purpose parallel applications.

As individual threads take divergent execution paths, they are processed sequentially. This serialization defeats part of the performance advantage of SIMD execution. Brunie [7] presented two complementary techniques that mitigate the impact of thread divergence on SIMD micro-architectures. Both techniques relax the SIMD execution model by allowing two distinct instructions to be scheduled to disjoint subsets of the the same row of execution units, instead of one single instruction. They increase flexibility by providing more thread grouping opportunities than SIMD, while preserving the affinity between threads to avoid introducing extra memory divergence.

To overcome the limitations of the SIMD model when running control flow intensive programs on a GPGPU hardware, Malits [42] propose to use hierarchical warp scheduling and global warps reconstruction, implementing an ideal hierarchical warp scheduling mechanism that they call ODGS (Oracle Dynamic Global Scheduling) designed to maximize machine utilization via global warp reconstruction. Many general purpose data parallel applications are characterized as having intensive control flow

and unpredictable memory access patterns. Optimizing the code in such problems for current hardware is often ineffective and even impractical since it exhibits low hardware utilization leading to relatively low performance. They showed both analytically and by simulations of various benchmarks that local thread scheduling has inherent limitations when dealing with applications that have high rate of branch divergence.

## 2.7 Previous study of data redundancy in parallel applications

In our context, data redundancy is the commonality that exists between data manipulated by different threads. Data redundancy tends to be increased by good thread synchronization heuristics, as we show empirically. There have been different attempts, in the literature, to describe ways to capitalize on data redundancy. We review some of these works in this section.

Data locality is an important factor in the development of high-performance programs. The literature traditionally considers two types of locality in sequential applications: spatial and temporal. Recently, Meng *et al.* [43] have introduced the notion of inter-thread locality in the context of the SIMD execution model. If two separate threads simultaneously read data from nearby memory cells, then these accesses are said to have good inter-thread locality. In this case, a single memory access might provide data to several different threads. The importance of inter-thread locality is clear in the context of graphics processing units, given that memory access coalescing is, according to many authors, the most important optimization in this environment [56, 70, 34].

Collange [12] presented a hardware mechanism which dynamically detects uniform and affine vectors used in GPGPU applications. Collange's technique minimizes pressure on the register file and reduces power consumption with minimal architectural modifications. The vector units execute the same instruction on the same data leading to as many unnecessary operations as the length of the vector when uniform data are encountered. These unnecessary operations involve data transfers and activity in functional units that consume power. These transfers are a critical concern in architectural and microarchitectural designs of GPUs. This optimization can benefit up to 34% of register file reads and 22% of the computations of GPGPU applications.

Later on, Collange [13] proposed the Affine Vector Cache, a compressed cache design that complements the first level cache. Preserving memory locality is a major issue in highly-multithreaded architectures such as GPUs. As each thread needs

to maintain a private working set, all threads collectively put tremendous pressure on on-chip memory arrays, at significant cost in area and power. He showed that thread-private data in GPU-like implicit SIMD architectures can be compressed by a factor up to 16 by taking advantage of correlations between values held by different threads. It resulted in a global performance increase of 5.7% along with an energy reduction of 11% for a negligible hardware cost. The AVC stores blocks of memory that obey specific patterns named affine vectors. Affine vectors consist of either a sequence of equal integers, or a sequence of uniformly-increasing integers.

Finally, Long [39] studied fetch-identical and execute-identical instructions, which we explain in section 1.2. The fetch-identical instructions are important to synchronize SIMD threads and to increase the performance. The execute-identical instructions is a case of inter-thread data redundancy in which the instructions executes the same computation. He proposed methods to detect execute-identical instructions and to avoid multiple executions of the same computation.

## 2.8 Final considerations

We conclude this chapter justifying our work. We believe that this dissertation is useful to computer architects and compiler writers because it presents the first comparative study between such a vast number of different synchronization heuristics. Additionally, there is space for innovation in the wide horizon of reconvergence algorithms. Nobody has proposed a relevant mechanism to synchronize threads running on programs with complex function call graphs, for instance. In particular, from the survey in this section, we conclude that the research community has ignored function calls when designing thread synchronization algorithms. As an example of this omission, we have a clear need of a method to reconverge the execution flow before different threads enter the same function called from divergent program points.





# Chapter 3

## The heuristics

As we had said, a heuristic is a policy that the hardware use to schedule the threads. Each thread has its own PC. The heuristic is executed at the moment just before each fetch operation. It chooses the PC to be fetched and all threads with the chosen PC will execute an instruction. Threads with different PCs cannot execute simultaneously on the SIMD hardware.

It is expensive to execute a heuristic before each fetch, but most heuristics can be executed to choose the PC whenever the program finishes a basic block, calls a functions or return from a function. Thus, the chosen threads with the same PC will execute the entire basic block simultaneously. This solution improves the performance, but does not change the synchronization capability.

In this section we present the heuristics and explain how they work. The hardware implementation is not our goal, because some heuristics are too expensive. However we included the complexity analysis of some heuristics. This complexity is given in terms of the delay caused by its execution and the area of the hardware implementation. In the analysis, we will consider  $n$  as the number of threads.

### 3.1 Hardware-level heuristics

These heuristics affects only the thread scheduling that is done by hardware. They do not propose changes in the ISA (Instruction-Set Architecture) and have to be independent from the compiler output for the architecture. They have the advantage of being compatible to programs compiled before the hardware implementation.

The main problem of these heuristics is the lack of hardware support to identify the end of a basic block. Terminator instructions allow to detect the end of a basic block. These instructions are conditional or unconditional branches. However a

basic block may finish with any instruction, because the next instruction may be the destination of a branch, and the end of the basic block cannot be detected by the hardware. Another problem is that the instructions *call* and *return* may be used with purpose not related to functions in general programs.

A new ISA, tailored to this SIMD world could give the hardware the tools to know where each basic block begins and ends. Notice, however, that obtaining this information is still possible without changes in ISA using the compiler. When a basic block ends with a non-terminator instruction, the compiler can insert an unconditional jump instruction at the end of the basic block. This jump is to the next instruction to execute, the first instruction of the next basic block. But the heuristic will not work with older programs compiled without the compiler solution.

### 3.1.1 Minimum-PC

This heuristic was proposed by Quinn [53]. It is one of the first synchronization methods and exists since the late 80's. This is a simple synchronization method that was proposed to reconverge SIMD programs in distributed programs and nowadays it is used as base to synchronize threads in GPU architectures [19, 36].

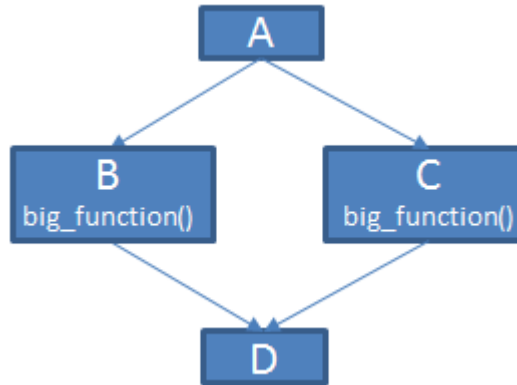
This heuristic follows the idea that a function starts with the smallest PC (program counter) and ends with the largest PC. Then, threads with smallest PC are behind and needs to execute. It works by giving priority to the thread with the smallest PC.

This heuristic does not take into account any other details about the state of the machine and does not deal with function calls in any special way. Consequently this heuristic does not perform well with function calls. However in some cases, it is able to overcome better heuristics if the address of the functions code are ordered favorably by the linker. To achieve good results, it requires the compiler and linker to statically lay out the binary code of each function according to the function call graph, complicating the build process and preventing its use on existing applications.

This heuristics requires to find the thread with the smallest PC value among all threads. Thus, in a sequential machine, its time complexity would be  $O(n)$ . Yet, on a hardware implementation this action can be implemented using parallel comparison with reduction tree. The delay complexity order is  $O(\log n)$  and the area complexity order is  $O(n)$ .

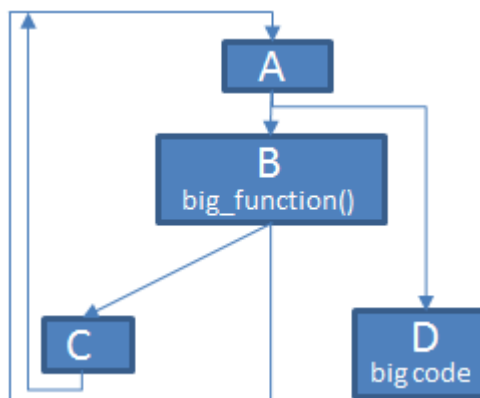
The behavior related to function calls when this heuristic is used can be seen in some of our tests. There are cases in which callee functions with smaller PC is favorable, as shown in the CFG in figure 3.4. But there are cases in which callee functions with

larger PC is favorable, and it is shown in figures 3.1 and 3.2. Therefore, it would be difficult to use the linker to sort the functions favorably.



**Figure 3.1.** CFG of if-then-else with redundant code

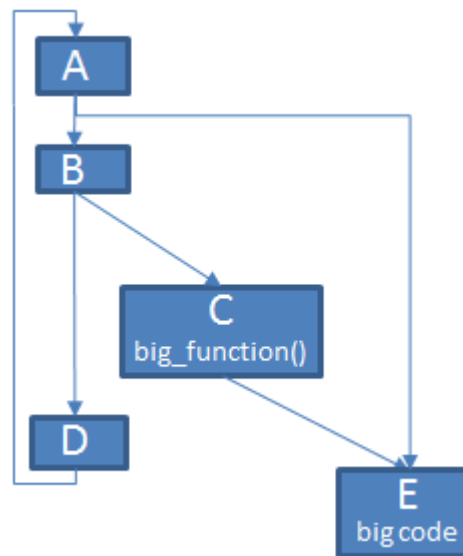
In figure 3.1, if the PC of the big function is larger than the PC of the calling function, the Min-PC heuristic will be able to synchronize the execution of the big function. A thread will execute the ‘then’ and the other will execute the ‘else’. The ‘then’ block will be executed firstly and it will call the big function. The function has the PC larger than the PC of the ‘else’ block, then the current thread will wait and the other thread will execute the ‘else’ block and call the same big function. Thus, the execution of the big function will be done simultaneously by all the threads in flight. However, if the PC of the big function is smaller than the PC of the calling function, this heuristic will not be able to do this synchronization.



**Figure 3.2.** CFG of loop with continue statement and big code in the beginning

In figure 3.2, a good synchronization happens if the address where the big

function starts is larger than the address of the calling function and the number of iterations of all threads is the same. Thus, if a thread that executed the ‘continue’ statement call the function and there are threads that did not execute ‘continue’, then the calling thread will wait until the other threads call the function. Consequently, the beginning of the iterations of the loop will be synchronized. When the address of the big function is not favorable, the threads that did not execute the ‘continue’ statement will not execute the big function simultaneously with the threads that did.



**Figure 3.3.** CFG of loop that may end with break statement

In figure 3.3, the test is a loop that calls a big function, if it stops through the break statement. With Min-PC, if the PC of the big function is larger than the PC of the calling function, then the threads that exit through break will execute the big function simultaneously. It does not happen if the PC of the big function is smaller. Finally, if the PC of the big function is larger, the threads that exit through the main way will continue without waiting the threads that called the big function.

### 3.1.2 Minimum SP, Minimum PC

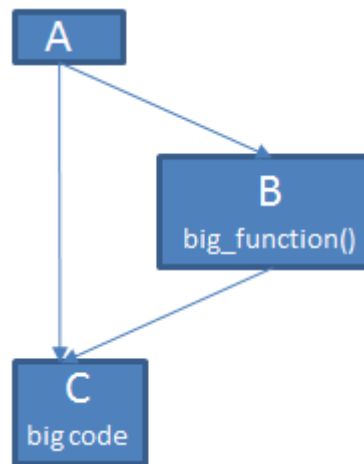
Collange [11] idealized MinSP-MinPC and we implemented and tested it [44]. This heuristic improves Quinn’s Min-PC heuristic in programs that contain function calls. Any function calls are supported, including indirect calls and recursive functions. The address where the called function begins may be anywhere in the program and it may mess with the choice of the smallest PC.

This heuristic follows the idea that threads with smallest SP (stack pointer) called functions and they have not returned yet. The call stack grows down, therefore the SP value decreases when it points to larger stacks. Knowing this, this heuristic chooses the threads with the smallest SP. When there are multiples threads with the same smallest SP, it chooses the threads among them with the smallest PC to execute.

Despite this heuristic being good with recursive calls, it still cannot synchronize the threads when the same function is reached by different threads through different intermediate functions called by indirect function calls. In compiler analysis jargon, this heuristics is 1-Context Sensitive. In other words, it works considering only one nesting depth in the activation stack of functions.

Although this heuristics is an improvement on Minimum PC, it also presents shortcomings. This heuristic requires the SP register, that may be not available in some architectures. It also requires that programs do not use the SP for other purposes. Instructions that manage the stack, such as push and pop, may compromise the effectiveness of the heuristics.

The complexity of this heuristics, in terms of space and area, is the same as in Min-PC heuristic.



**Figure 3.4.** CFG of if without else block that calls a big function

The figure 3.4 shows the improvement of MinSP-MinPC over Min-PC. If the PC of the called function is smaller, then Min-PC will be able to synchronize the caller and the callee threads. But if the PC is larger, then the long code after the ‘if’ block will be executed twice, because the function will be executed after the thread that did not call it execute the long code. This example justifies the use of the heuristic MinSP-MinPC, because it does not depend on the PC of the called function.

### 3.1.3 Maximum Function Level, Minimum PC

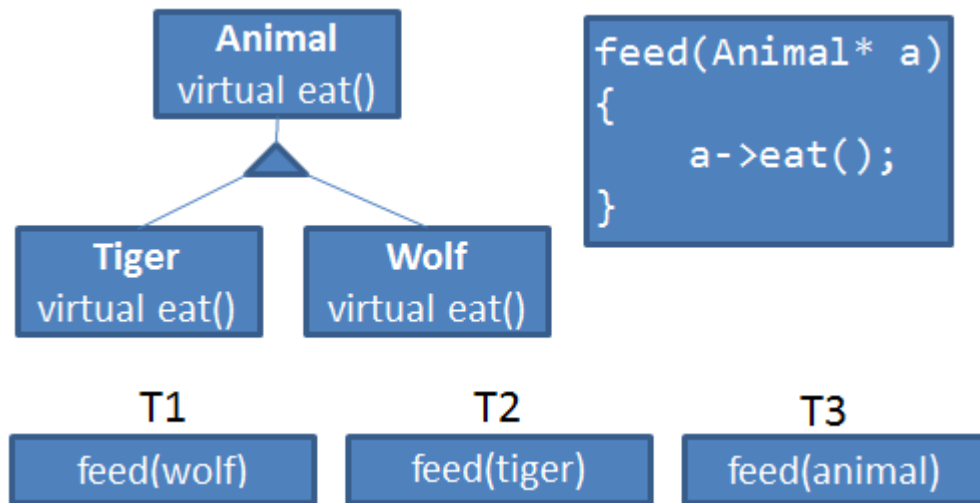
MaxFun-MinPC (Maximum Function Level - Minimum PC) heuristic is much like MinSP-MinPC, but instead of choosing the smallest SP, it chooses the thread that has the the highest number of activation records on the stack. Therefore, this heuristic would have the same behavior of MinSP-MinPC if all activation records had the same size. This idea is based on the counters that POMP machine uses and similar idea is used in GPUs.

This heuristic can be used in architectures in which the SP register is not available. Each thread would have a register that keeps the current number of activation records in the stack. This heuristic can use the *call* instruction to increment the function call depth and the *return* instruction to decrement it. However it will not work if the program uses the instructions *call* and *return* for other purposes, which are not related to functions. It will also not work if the functions are called and returned by unusual ways, because the heuristic requires a return for each call and vice versa. Therefore it does not work if the C *longjump* is used or exceptions are thrown.

In order to support exceptions, the heuristic would require compiler support to execute an useless return instruction for each function that is interrupted by the exception. This return instruction has the purpose of decrementing the function depth value. It would be executed like C++ destructors for all functions of the program. Although this heuristic requires compiler support to deal with exceptions, it does not require changes in the ISA.

The complexity of this heuristics, in terms of space and area, is the same as in Min-PC heuristic.

In figure 3.5, we have a class with derivated classes and both implementations of *eat* call the overridden function in the parent class. If the indirect call leads a thread to *Wolf::eat* and another thread directly to *Animal::eat*, the thread that called the derivated class should be executed first to synchronize the execution of *eat* in the parent class. With MinPC heuristic, if the address of *Wolf::eat* is smaller than the address *Animal::eat*, this synchronization will be possible. MinSP-MinPC heuristic will depend on the size of the activation record of *Wolf::eat* and *Animal::eat*, if the size of the former is larger, and therefore having the smallest SP, *Wolf::eat* will be executed firstly and the synchronization will happen.



**Figure 3.5.** Example of a virtual function

We implemented three versions of this heuristic to decide what to do when an indirect call is executed and the threads start different functions. In this case, the function level will be the same in all threads. Each version has a priority to choose a thread, before choosing by the smallest PC.

1. Nothing.
2. Choose the thread with largest SP, in other words, the function with smallest activation record.
3. Choose the thread with smallest SP, in other words, the function with largest activation record.

With the example in figure 3.5, first implementation has the same behavior as MinPC, and the third implementation has the same behavior as MinSP-MinPC. The second implementation will be able to synchronize the function if the size of the activation record of *Wolf::eat* is smaller. In the end, we do not have a rule that is the best for any case.

### 3.1.4 Long

Long [39] created this heuristic to analyse redundancies in SIMD programs. Its key idea is to add some sort of *memory* to each thread. One thread uses the memory of the others to advance or stall. If the current PC of a thread  $t_0$  is in the recent history

of another thread  $t_1$ , then thread  $t_0$  is probably behind  $t_1$ . In this case,  $t_0$  needs to progress to catch up with  $t_1$ .

This heuristic uses a table called FHB (Fetch History Buffer), which is a circular buffer where each thread will store the PC of the last basic blocks that it executed. Whenever a basic block starts, its PC, which is the PC of its first instruction, is inserted into the table. This table has a finite number of spots. Thus, an update may cause the dropping of the oldest entry.

We consider the code just after a call instruction as a new basic block, in other words, the instruction after *return* is being considered as a new basic block. In our implementation, this heuristic works ignoring basic blocks that are not easily detectable. A basic block is not easily detectable if it does not start after a non-terminator instruction. However this heuristic can be implemented with changes in ISA or compiler support to allow the detection of the beginning of any basic block.

The heuristic works using the FHB to take choices. Each thread receive 1 point of priority for each thread whose FHB contains the basic block that the current thread is executing. The threads with highest priority will be chosen to execute, but threads with different PC and the same highest priority execute each entire basic blocks alternately.

The main problem of this heuristic is that it is very expensive to be implemented in hardware. This difficulty stems from the fact that it needs to keep tables with data. Maintaining and consulting these tables is expensive. Therefore, this heuristic should be used only in simulations. It could be used, for instance, to find instruction redundancy in emulated programs.

Long's heuristic has the size of the FHB table as an implementation parameter. According to the results of our experiments, which are presented in chapter 5, no specific size is the best option always. The sizes of the table that we are using in this dissertation are 2, 4, 8, 16, 32 and 64. Higher values are expensive and we observe, empirically, that the result is not good. So far, we have not been able to provide a rationale on why this is true.

The required memory to store the FHB table for all threads is also  $O(n*s)$ , where  $s$  is the size of the FHB table. The delay complexity order is  $O(\log n)$ , because each thread check if its PC is in the table of other threads parallelly and the number of occurrence is compared. The energy consumption is high, because the number of parallel consults in FHB is  $O(n*n)$ .

Figure 3.1 shows a situation in which this heuristic is good. When the threads diverges, the threads are executed in a round-robin fashion. Long's heuristic is able to detect when multiple threads are executing the same function and it is able to synchronize the execution of this function.



And figure 3.4 gives a case in which this heuristic is bad. With the thread divergence, a thread will execute the ‘if’ block and other thread will continue the execution after the block. This heuristic is not able to detect ‘if’ statements without ‘else’ and fails. Therefore, the basic block C will not be synchronized and will be executed twice.

### 3.1.5 Variations of Long’s Heuristic

The idea of Long’s heuristic can be used to create other heuristics. When multiple threads have the same highest priority, the original heuristic executes these threads alternately, but the variations of Long’s heuristic use other policies to choose the next thread to execute. During this research, we have proposed and experimented three variations of Long’s method, which we explain in this section:

- **Long’s with Min-PC:** When multiple threads have the highest priority, the basic block of the threads with smallest PC is executed firstly. Then, all priorities are recounted again for the next choice.
- **Long’s with MinSP-MinPC:** This is the same as above, but uses MinSP-MinPC policy.
- **Long IRB (Instruction Round-Robin):** This is almost the same as original heuristic, but it stores the address of each executed instruction instead of the beginning of each basic block. It schedules single instructions instead of basic blocks. It is able to expose more TLP than the original heuristic.

### 3.1.6 Lee

This heuristic was proposed by Lee [36]. It uses two lists of threads, *current* list and *future* list. The heuristic chooses the threads with the smallest PC from the current list. When an conditional or unconditional branch is executed, if the PC of a thread has the value changed to a smaller address, the thread is removed from the current list and inserted into the future list. When the current list becomes empty, both lists are swapped.

The complexity of this heuristics, in terms of space and area, is the same as in Min-PC heuristic. But it is more expensive, because it uses two lists of threads and chooses a thread from one of the lists.

While this heuristics optimizes loops with *continue* statements, which Min-PC is not good to execute, it worsens loops with *break* statements. In figure 3.2, an example

of loop with ‘continue’, the threads that execute ‘continue’ will be placed in future list and will be delayed. When the other threads finishes the iteration, all threads will execute together. However, if the threads execute a different number of iterations, the threads that finished the loop will continue and the threads that started the next iteration will be waiting in the future list.

Figure 3.3 is an example of loop with ‘break’. The big code will be executed twice, because ‘break’ is not a backward jump. The threads that starts a new iteration will execute a backward jump and consequently they will be placed in future list and will be stopped.

## 3.2 Heuristics that require ISA change or compiler support

The heuristics that we have described previously can be implemented at the hardware level. This mean that they can be used on binaries already deployed. However, if we are allowed to re-compile programs, than we can do even better. In this section we describe heuristics that demand compiler support. Some of them also require changes in the ISA.

These heuristics require that the compiler analyses and modifies the baseline program. The changes in the ISA include the insertion of data from the compiler static analysis and insertion of new instructions, for instance, insertion of synchronization barriers in the code. In this case, the compiler can analyse the code to decide where the barrier should be inserted. The compiler can also change the control flow graph of the program to optimize the heuristic or meet its requirements.

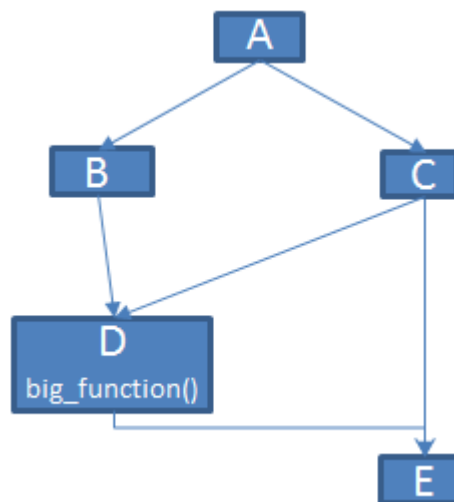
The drawback of this group of heuristics is that programs that were compiled before the change in ISA cannot run in the hardware. The new version of the hardware may use a new and better heuristic that is not compatible with older programs. In order to allow the execution of older programs, the hardware has to keep the implementation of the older heuristic and it is expensive. However we notice that changing the ISA is not a problem in the GPGPU context, because languages like OpenCL foster a runtime environment that keeps the source code in the program distribution and this source code is compiled just when the GPU that will execute the code becomes known.

### 3.2.1 Nearest post-dominator reconvergence

The reconvergence in post-dominators, proposed by Fung [22], uses the compiler to find the nearest post-dominator basic block of each branch in a function. We say that a point  $p$  in a directed graph with an ending node  $e$  post-dominates another point  $v$ , if every path from  $v$  to the end of the graph  $e$  must go across  $p$ . The post-dominator of a branch is the point where the control flow will reconverge certainly. In other words, it is a point that will be always executed, independent on the result of the branch. Furthermore, the post-dominator of the branch is the "nearest" point to be always executed. When a group of active threads execute the branch and the threads diverge, all threads that reach the post-dominator will wait until all the other threads yet to reach that place make progress. Thus all threads will be synchronized again at the post-dominator of the branch.

This idea requires a helper heuristic to execute divergent threads while the post-dominator is not reached. Many helper heuristics can be used:

- Minimum PC (Quinn's)
- Minimum SP, Minimum PC
- Maximum function level, Minimum PC
- Maximum SP, Minimum PC
- Minimum function level, Minimum PC
- Round-robin



**Figure 3.6.** CFG with distant post-dominator

The helper heuristic has an important impact on the effectiveness of the core algorithm. Fung has proposed a heuristic that executes each divergent path separately and the result was not good. In the CFG in figure 3.6, A is a branch and its post-dominator is E. If two divergent threads will execute respectively (A, B, D, E) and (A, C, D, E), D will be executed by both threads. The post-dominator idea alone will not be able to make both threads execute D simultaneously, because D is not a post-dominator. However, if the Min-PC is used as a helper heuristic, it will be able to synchronize the threads before the execution of D starts.

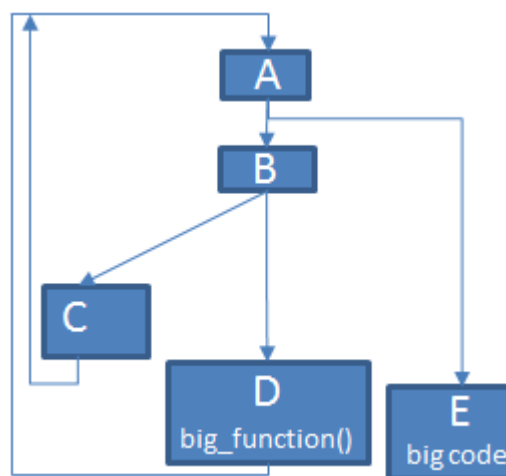
Our implementation of this heuristic is complex, because it does not put limitations on the program, such as forbidding nested divergent structures. Our implementation is:

- The pair branch and its nearest post-dominator have an identifier that is unique in the function.
- At compile-time, the compiler has to put a mark in each branch and its nearest post-dominator. The marks will be read by the heuristic at run-time.
- The hardware has to use a stack or buffer to store each unreconverged divergence. The highest size of the stack is the number of threads, because it is the highest number of divergences. It is needed to support programs with nested structures of branch and its post-dominator.
- When a divergence happens, the hardware must push the identifier of the branch to the stack along with the mask of active threads. Indirect function calls are also divergent points and its post-dominator is the next instruction after the return of the function call.
- If the branch identifier is already on the stack and the thread finds it again, then the branch should be ignored. It happens in loops when the ending condition is tested.
- When a thread finds a post-dominator mark, it searches for its identifier in the buffer to know what are the threads that executed the corresponding branch. Then, it blocks the thread until all threads reach the post-dominator.
- A thread may find a post-dominator mark that belongs to a branch that it did not execute or did not diverge. In this case, the thread has to ignore the wait in the post-dominator.

- The same point may contain multiple post-dominator marks, because it may be the post-dominator of multiple branches.
- The function level (the number of activation records) has to be stored along with the branch identifier. The same function may be on the call stack and it happens with recursive functions.
  - If the same branch is found again, but the function level is higher, then it can be pushed in the stack again.
  - If a post-dominator is found, but no corresponding branch with the same level is found in the buffer, then the post-dominator has to be ignored. Branches with the same identifier and lower function level can be on the buffer, but has to be ignored.

The iterations of a loop with *break* statement, as in figure 3.3, cannot be synchronized by this heuristic, because the post-dominator of the branch that leads to the *break* statement is after the loop. The behavior of the heuristic will depend on the helper heuristic.

This heuristic is able to synchronize the iterations of the loop with ‘continue’, as in figure 3.2. In the examples, the iteration starts with a big code and execute a branch to decide if the program will continue executing the current iteration or will start the next iteration. In this example, Min-PC will start the next iteration of the threads that executed ‘continue’ and the threads that did not will be stopped and will not execute the big code. With this heuristic, the threads that executed ‘continue’ will wait the threads that have not finished the iteration



**Figure 3.7.** CFG of loop with continue statement and big code in the end

In figure 3.7, we see the opposite, because the synchronized iterations have not good results. It has a big code after a ‘continue’ statement. Let’s use the case where we have two threads and two iterations, and only the first thread executes ‘continue’ on the first iteration and only the second thread executes ‘continue’ on the second iteration. In this case, the big chunk of code will be executed twice. Min-PC does not suffer this penalty, for instance. If we use this heuristic, the first thread will wait the second thread to start the next iteration and both threads will execute the big code together.

Figure 3.1, is an example in which if we use a helper heuristic that chooses the threads with the largest SP or smallest function level, the execution of the big function will be synchronized. When a thread call a function, the threads that did not call it will execute. If there were a third block, such as ‘else if’, the post-dominator would avoid that the threads continue after the post-dominator without waiting the other threads.

### 3.2.2 Heuristic with structured control flow graph

Programs with structured control flow graph are easier to synchronize. Notice, however, that many compiled programs do not have structured control flow graphs. Even though there are ways to convert a non-structure program to a structure format, these approaches have some shortcomings [71, 69]. In particular, they suffer from code expansion. Nevertheless, we will describe a synchronization heuristic that assumes structured code in this section.

The POMP machine, which uses a register for each thread to count the depth of control flow structures, can be improved. Support to indirect function calls can be added easily to execute each different function per time. Another improvement is the inclusion of explicit instructions that represents the *continue* and *break* statements with any nesting depth. It allows to finish an iteration or the loop from a deep nesting level.

With the support of *break* and *continue* instructions, an algorithm to transform unstructured CFG would create structured CFG that uses these instructions. Consequently it would have smaller code expansion in comparison to the original transformation method, because it would have more options and do less changes in the code.

Due to the code expansion, in order to compare the performance of this heuristic, we need to check if the resulting sequence of instructions executed by a program using this heuristic is smaller than the result of another heuristic without code expansion.

Code requirements:

- All loops must have the structure "while(true)...".
  - The end of an iteration is the instructions 'continue' (unconditional) or 'if-continue' (conditional).
  - The end of the loop is the instructions 'break' (unconditional) or 'if-break' (conditional).
  - "while(cond)..." should become "while(true) if(!cond)break; ... continue;"
  - "do-while(cond)..." should become "while(true) ... if(cond)continue; break;"
- No indirect branch is allowed, which includes *switch* statements.

How it works:

- Each thread has its nesting depth register (NDR).
- Threads with the highest NDR are chosen to execute and the heuristic must ensure the same PC.
- Special instructions changes the values of NDR and reschedule threads.

Added and modified instructions:

- if-then: This is a simple 'if' statement without 'else'. The NDR of the threads that enter in the block is incremented when the block starts and decremented when it ends.
- if-then-else: This is an 'if' block with 'else'. Firstly it executes the 'then' block. When it ends, it executes the 'else' block. When it ends, the threads will be synchronized.
- call: This call a function and the NDR is incremented. Recursive functions are supported. Indirect calls are also supported, each different function would have a different increment in NDR and the value of NDR to decrement is saved on the call stack.
- return: This instruction can be placed anywhere. It returns and decrements the NDR by the number of nesting depths it increased since the function started and this number is statically known as parameter of the instruction.

- **break:** This ends the current loop, decrements the NDR by the desired nesting depth that is statically known as parameter of the instruction and sends the PC to after the broken loops.
- **continue:** This ends the current iteration, decrements the NDR by the desired nesting depth that is statically known as parameter of the instruction and sends the PC to the beginning of the desired loop.
- **if-continue:** This is a conditional ‘continue’ and have no intermediate basic block.
- **if-break:** This is a conditional ‘continue’ and have no intermediate basic block.
- **begin-while:** This is a neutral instruction that increments the NDR before the beginning of the loop. This is the beginning of each iteration and the destination of ‘continue’ instructions.
- **end-while:** This is a pseudo-instruction at the end of the loops. It is an unconditional jump to go to the next iteration. It is really the instruction ‘continue’.
- **reconvergence:** It is a neutral instruction placed in the end of ‘then’ blocks that are without else and in the end of any ‘else’ blocks.

The implementation can be Improved:

- When an ‘if’ block that is inside a loop and it ends with ‘break’ or ‘return’, the execution of this basic block should be delayed until all threads finish the loop or achieve this basic block. Thus, all thread can execute this block simultaneously. A neutral instruction should be placed in the beginning of the block saying to wait.
- There are cases where ‘then’ and ‘else’ call the same function, such as in figure 3.1, and we would like to synchronize the execution of the function. The solution could be branch merging or a neutral instruction asking to wait when the function is called.

### 3.2.3 Thread frontiers

The thread frontiers [19] method uses the compiler to give priority to basic blocks following a policy. During the execution, the hardware chooses to execute the threads in basic block with the largest priority. The challenge is to find the best priority policy



to use and multiple options are possible. For instance, MinPC can be a priority policy, in this case, the compiler can give priorities based on the order of the basic blocks in the original program. The policy is hardware-independent, because the priorities are assigned at compile-time. Thus, the only change in ISA is the support to follow the policy and the compiler can choose any policy.

The main priority policy is to give higher priority to basic blocks whose longest distance in number of basic blocks in the CFG until the end of the function is larger. The longest path until the end of the function cannot include repeated basic blocks. This policy cannot be used with large CFG, because the longest path problem is NP-hard [58]. Nevertheless, if the graph is acyclic, in other words, the graph does not have any loop, a solution in polynomial time for this case exist. The reference implementation of thread frontiers is in Ocelot [20], a dynamic compilation framework used for GPGPU. Its priority policy uses an algorithm based on edge-covering tree, which is a heuristic to find an approximated solution for the longest path problem.

This policy of longest path includes the post-dominator features. Any basic block that comes before its post-dominator has larger priority than its post-dominator, because the post-dominator is ever included in any path until the end of the function. Therefore, its longest path until the end includes the post-dominator. Furthermore, thread frontiers is much cheaper than a full implementation of the post-dominator heuristics. We could discard post-dominators heuristics, however the post-dominator idea can still be used to try different approaches with some sub-heuristics, such as MinFun-MinPC, MaxSP-MinPC and Round-robin, which cannot be done with thread frontiers.

In figure 3.2, we can compare Min-PC, thread frontiers and generic post-dominator heuristics. If a divergence happened in B, a thread will execute C and the other thread will execute D. C and D should be executed before A to synchronize the threads. Min-PC executes C followed by A firstly, because C and A have PC smaller than D. Post-dominator executes A and D firstly, because A is post-dominator of the branch in B. And thread frontiers will execute C and D before A, because the longest path from C and D until the end of the function, without repetition, is larger. The distances are: A is 1, B is 3, C is 2, and D is 2.

This method can be implemented without changes in ISA. The compiler can transform the CFG of the function by sorting the basic blocks by basic blocks with largest priority. Thus the basic block with larger priority would have smaller PC and consequently we can use the Min-PC heuristic in hardware for intra-function synchronization. The PC is available in the hardware, but the priority would have to be presented in the beginning of the code of each basic block. In figure 3.3, if the

compiler moves the basic block C to the place between D and E, the PC of C will be larger than the PC of D and smaller than the PC of E, and Min-PC will be able to synchronize it correctly.

This method does not specify how to handle function calls, but allows the help of specific techniques. Most GPGPU heuristics use a stack or counters of activation record per thread in order to make the threads that called a function have higher priority. Thus, this heuristic can have the benefits that MaxFun-MinPC has related to function calls.

We implemented two versions of this heuristic using the policy of the longest of the longest path until the end. The difference is the behavior when the priority of different basic blocks is the same. In the first version, we choose the thread by smallest PC. And in the second, it executes each basic block with the same priority in a round-robin way before choosing the next basic blocks.

### 3.3 Experimental Heuristics

These are heuristics created by Collange, as explained in section 1.5.1, to be used in experiments. They have not been published yet. Some of them may be expensive to be implemented in hardware, because they may need to store data about the program for each basic block. Other of them may be impossible to implement in hardware when they check the future by reading the execution traces. Their main goal is to reach higher DLP or TLP to help us to know that it is possible to have better results, which other heuristics cannot reach.

#### 3.3.1 Reconvergence Detection

This is an experimental heuristic that does not produce good results. It uses a mask in the beginning of each basic block to know what threads that were executing the basic block previously to detect when a reconvergence was missed. When a group of thread executes a basic block, its mask is saved. If a basic block is executed by a group of thread whose mask is not the same as the previous mask, then the active threads will be blocked until the threads that were in the previous mask achieve this basic block. This heuristic is very expensive to be implemented in hardware, because it keeps a mask of the previous active threads in each basic block.

### 3.3.2 Distance

This is an experimental heuristic that is expensive to be implemented in hardware. It takes some ideas from Reconvergence Detection heuristic and improves them. It tries to detect the divergence points and its corresponding reconvergence point. It estimates dynamically branch lengths. It is able to output high TLP when high DLP is not possible. Common heuristics do not know if a branch instruction is an ‘if’ block with or without an ‘else’ block. This information is important to decide if the two destinations of branch should be executed simultaneously in opportunistic SIMD to output high TLP without worsening in DLP and it cannot be done with ‘if’ statements without ‘else’ blocks.

### 3.3.3 Greedy Oracle

This is a heuristic for testing purposes. It cannot be implemented in hardware, because it sees the future, by accessing the execution trace. It sees the next instructions of the execution trace before taking decision. It checks the future of each thread that is not active to find the PC of the active threads. The priority is given according to the distance until the found PC. Large distance has higher priority. The heuristic has as parameter the number of future instructions to check. High numbers have slower performance. If the PC is not found, the highest priority is given.

### 3.3.4 Round-Robin

This heuristic was used to simulate the execution of the program in a true MIMD machine. Thus it has the maximum TLP and minimum DLP. It is not used for SIMD and opportunistic SIMD because it does not share instructions. It is implemented to choose only a single instruction of a thread per time to execute. The next chosen instruction belongs to another thread and after all threads execute an instruction, it starts again with the first thread.

A variant of this heuristic is Round-Robin-Eq. This heuristic works like Round-Robin, but it executes the threads with the same PC simultaneously whenever it is possible, unlike Round-Robin. If the PC of all threads are different, this heuristic has the same behavior as Round-Robin. Thus, this heuristic is able to share instructions and have higher DLP.



# Chapter 4

## Methodology

As we have mentioned, one of the main contributions of this work is an extensive comparison between different thread synchronization heuristics. This comparison uses well-known benchmarks, and is implemented on top of an industrial-quality infrastructure. In this section we describe these benchmarks, and the testing infrastructure.

### 4.1 Benchmarks

One of the main difficulties that we faced when working on this project was related to the choice of benchmarks. To be able to compare the different heuristics in a meaningful setup, we have defined a few properties that should be present in each benchmark:

- Data-parallelism. Multiple threads execute the same program for different data.
- Balanced load distribution between threads. In other words, the number of instructions that each thread execute is likely to be similar, or has the potential to be similar.
- All threads must belong to the same process (multithreaded programs rather than multiple processes). We want the address space of the multiple threads to be the same. Thus, the PC of instructions will be the same for all threads, including instructions in shared libraries. This setup lets us study the patterns of memory access.
- Pthread implementation. Pthreads are lightweight compared to the more complex frameworks. Furthermore, pthread based programs are easy to instrument with our tools.

- No complex synchronization techniques. Benchmarks with *mutex* or *monitor* are accepted, because we can ignore the protected instructions by detecting where these instructions start and finish. However, *semaphore* is not allowed, because it does not protect simply a limited area that we can detect and ignore.

We chose the following benchmarks, which meet the requirements enumerated above:

- 4 benchmarks from **Parsec** [5] package:
  - **blackscholes**: option pricing with Black-Scholes Partial Differential Equation (PDE).
  - **swaptions**: application that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions.
  - **bodytrack**: computer vision application that tracks a human body with multiple cameras through an image sequence.
  - **fluidanimate**: fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method.
- 7 benchmarks from **Splash2** [66] package:
  - **barnes**: simulation of the interaction of a system of particles with the Barnes-Hut method.
  - **fmm**: simulation of the interaction of a system of particles with a parallel adaptive Fast Multipole Method to simulate the interaction of a system of particles.
  - **fft**: signal processing application that uses Fast Fourier Transform.
  - **radix**: radix sort algorithm.
  - **volrend**: A raytracer algorithm.
  - **ocean\_ncp**: simulation of large-scale ocean movements based on eddy and boundary currents.
  - **water\_nsquared**: simulation of water molecular dynamics.
- **Tachyon** raytracer [62]

These benchmarks were implemented in C/C++. We compiled them to 64-bit x86 architecture using Clang 3.1 [35]. Therefore, whenever we mention *number of instructions* we mean *number of x86-64 instructions*. Some of the benchmarks had different

implementations. Whenever more than one implementation would be available, we chose that one which uses pthreads.

<b>Benchmark</b>	<b>Insts</b>	<b>Seq</b>	<b>Crit</b>
blackscholes	787	533	0
bodytrack	21,067	1,026	65537
fluidanimate	5,274	5,691	5856
swaptions	2,248	1,123	0
tachyon	5,796	542	1346
barnes	4,164	1,414	6131942
fft	2,004	697	2102
fmm	7,289	2,668	6749323
ocean_ncp	11,726	379	36040
radix	1,627	734	2299
volrend	5,014	182	20663
water_nsquared	6,413	254	82875

**Table 4.1.** Characteristics of the benchmarks.

In table 4.1, **Insts** means number of static X86 instructions in the benchmark, **Seq** means number of instructions (in millions) in the dynamic sequence produced with Pin, and **Crit** means number of instructions in critical sections.

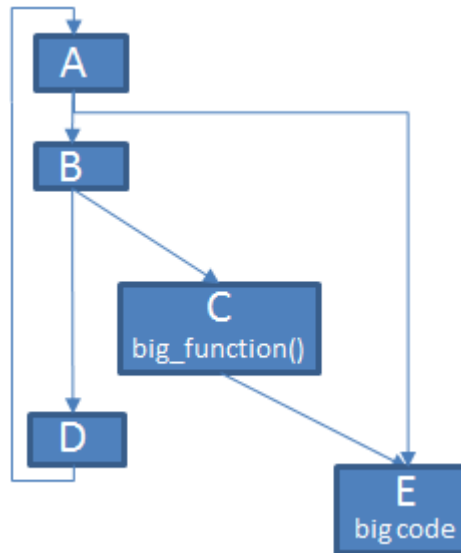
## 4.2 Execution

In order to execute the benchmarks and extract data, we have used the Pin [41] tool. Pin is an instrumentation framework developed by Intel. It is able to execute X86 programs and modify the code during the execution. It allows us to insert new instructions to extract information about the execution of programs. It can count instructions, detect function calls, detect memory access and can be used as debugger and as profiler.

In the work, we used Pin to test benchmarks and know if the benchmarks meet the desired requirements. We counted the number of instructions that each thread executes to check if the load is balanced between the threads. And we also used the instrumentation to know what is the first function that a thread execute. If the function is the same for all worker threads, the benchmark is data-parallel.

The main use of Pin was to create an execution trace for each thread of the benchmark. The execution traces contain the sequence of PC (instruction address) of each executed instruction. Each trace was stored in a file in disk. Along with the PC, we also report the value of the SP (stack pointer address) in the moment before the execution of the instruction. This value is changed to be relative to the bottom of the stack, which is the SP that we record when the thread starts. Thus, the SP that we obtain is the size of the call stack. We also include the value of all registers read by the instruction. If the instruction accesses memory, the memory address is also written.

As an example, we have a sample trace produced for the program seen in figure 4.1. If a thread executes the loop twice, for instance, the sequence of basic blocks in the trace would be: ABDABDAE. The traces contains the sequence of intructions of the basic blocks.



**Figure 4.1.** Example of trace

The traces generate output files with huge size that do not fit in the computer RAM, because the benchmarks have billions of instructions. Therefore we placed the trace files on the hard disk during the experiments. While the benchmark runs in a short time, it takes long time to generate traces. Furthermore, the heuristics take longer time to execute.

Local variables created during function calls are placed in allocation units called *activation records* and these records are stored in a space called the *stack frame*. When a memory access is done inside the stack frame, the target address is located between the SP and the beginning of the stack. Each thread has its own stack of data. In order



to observe the memory access patterns across threads, we would like that the target addresses in the function stack be equally spaced. We achieve this goal by separating the base address of each thread's stack by the same amount. We perform this conversion at instrumentation time. During the instrumentation, when stack accesses are detected, these addresses are modified by the correct offset, so that all the stacks are aligned.

To find more execute-identical instructions, all values of stack pointer (SP) and frame pointer (FP) were converted to be in relation to the base of the stack. However, we were not able to detect when the same arithmetic operation related to arrays were performed to find a address. The address of the array is different and we do not know if the arithmetic operation is to find the address of something in an array or to another purpose.

Our analysis works as follows: firstly we execute the instrumented benchmark using Pin. This instrumented program generates an execution trace, which we have described before. In a second phase, we go over these traces, simulating the application of each synchronization heuristic on it. During this simulation, we collect statistics, that we save to analyse later on. Thus, each heuristic will read the files that contains the sequence of instructions executed by each thread in flight. Each analysis produces three outputs. The first output is the total number of instructions fetched by the heuristic. The second output is the patterns of memory access. Finally, the third output is the percentage of fetch-identical instruction.

## 4.3 Instrumentation with compiler support

The heuristics that require compiler support work on modified binaries. In order to use these heuristics, we had to recompile the benchmarks, to augmented them with pragmas. In this work, pragmas are marks in the code and we use them to represent changes in architectures. The pragmas can be used, for instance, to represent synchronization barrier in the code.

We have used LLVM [35] to perform the modification and the insertion of pragmas. LLVM is a set of softwares related to program compilation. It has a virtual machine whose instructions are an intermediate format. Programs in high-level languages can be compiled to LLVM bytecodes and can be optimized or modified through **passes**, which are plugins written for LLVM using LLVM API.

Thus, we have created an LLVM pass to find suitable points to insert pragmas. When executing the benchmarks, we ignored the pragmas. In other words, these marks have no special semantics, other than guiding the heuristics. The heuristics will read

the pragmas in the code and take decision based on them. We compiled the benchmarks with optimizations, such as `-O2`, to generate LLVM bytecode and combined all modules into a single file with LLVM bytecode. Our LLVM pass is executed with this file as the last step before generating the final machine code.

A pragma is implemented simply as a call to an empty function, which is inserted by the LLVM pass and detected through Pin tool. We can only insert pragmas in code that we can compile easily. Consequently, the instructions of shared libraries, such as the C library and the pthread library, are ignored. Ignoring shared libraries does not have impact on the purpose of the work, because the benchmark were executed in a way in which the shared libraries were not part of the program and, therefore, they do represent serial or synchronized parts.

An example of the use of pragma is present to help the post-dominator heuristics. We used LLVM to find each pair of branches and their nearest post-dominator. We inserted a pragma just before the branch instruction with an identifier, to tell that the next instruction is a branch. And we inserted a pragma with the same identifier before the first instruction of the post-dominator basic block. This is the synchronization point of the post-dominator heuristic.

# Chapter 5

## Results

In this chapter, we present the results of our experiments using the methodology introduced in Chapter 4. We present the results of each benchmark running them with 16 threads. The results show the amount of data regularity that we found with the heuristic. The data regularity and instruction regularity depend mostly on the input data that the benchmark receives. If the data is very regular, the program will follow the same control and process similar data.

The first part of this chapter shows the comparison of heuristics. The main conclusion that we take from these numbers is that there is not a heuristic that is clearly better than the others. This conclusion leads us to think that the best heuristic depends on the nature of the application, and on its input data. Nevertheless, we show that the heuristics that we have designed are very competitive with state-of-the-art approaches. In fact, in many cases, we could outperform very complex synchronization algorithms using our simple Min-PC approach.

In the second part of this chapter we study the average number of active threads per application. On top of this study, we analyse the number of execute-identical instruction found in each benchmark. These results seem to indicate that 30% of the instructions are execute-identical on average. This is a promising result, as it indicate that the hardware industry has an enormous opportunity to save resources by exploring similarities between the instructions given to different threads.

Finally, we close this chapter analysing the memory access patterns of our applications. From this analysis, we conclude that we have a significant amount of uniform and affine accesses. This is another fact that points towards the great opportunities available to resource sharing. Given that threads tend to process instructions with the same, or similar, data, much can be saved if the hardware is able to share among different threads the same results that they produce. The columns

that lacks bar in the charts means that there was no time that we had that number of active threads, consequently we do not have memory access patterns for that number.

To measure the amount of execute-identical instructions and analyse the memory access patterns, for each benchmark, we used the specific heuristic that has the highest DLP with the benchmark. The highest DLP implies the highest instruction sharing, which represents fetch-identical instructions. Execute-identical instructions and memory access regularity only happens between fetch-identical instructions.

## 5.1 Comparison Between the Effectiveness of the Heuristics

In this chapter, we present an extensive comparison between the different heuristics. We show the DLP (data-level parallelism) and the instruction throughput of the heuristics for each benchmark. Both metrics are explained in section 1.5.2. DLP is the average number of active threads per cycle with the threads executing the same instruction, but the data may be different. Instruction throughput is the average number of active threads per cycle that can be executed simultaneously, but the instructions may be different.

In normal SIMD architecture, the best heuristics are the ones with largest DLP, because SIMD is used for data parallelism. A high DLP indicates that different threads have been able to combine instructions often times, because the instructions are fetch-identical. Thus, a large DLP means higher speedup, because performance in SIMD hardware is measured by the number of fetched instructions.

In opportunistic SIMD architecture, the heuristics with higher instruction throughput have better performance. The definition of opportunistic SIMD machine has been given in Section 1.5.2. Higher instruction throughput means that the average number of active threads per cycle is higher. Consequently the performance is better because the program is executed with less cycles. And DLP in opportunistic SIMD indicates the amount of instruction fetch that can be shared.

The left chart shows DLP and the right one shows instruction throughput. In the end, we present the average of the result of each heuristics, which was calculated by arithmetic mean.

Figure 5.1 shows the reduced names that we used in the charts and tables to identify each benchmark. It also shows the heuristic that we chose to analyse execute-identical instructions and memory access patterns. And figure 5.2 shows the reduced names of the heuristics.

Small name	Benchmark	Chosen heuristic
barn	barnes	LongMinSP(02)
blacks	blackscholes	MinSP MinPC
body	bodytrack	MinSP MinPC
fft	fft	MinSP MinPC
fluid	fluidanimate	Lee
fmm	fmm	LongMinSP(8)
ocean	ocean_ncp	MinSP MinPC
radix	radix	MinSP MinPC
swapt	swaptions	LongMinSP(04)
tach	tachyon	MinSP MinPC
volr	volrend	Long(16)
water	water_nsquared	Dist(0 0 10k)

Table 5.1. Subtitles of benchmarks

Small name	Heuristic	Small name	Heuristic
Dist(0 0 10k)	Distance (0 0 10000)	LongMinSP(04)	Long MinSP PC (04)
Dist(1 1 -1)	Distance (1 1 -1)	LongMinSP(08)	Long MinSP PC (08)
Dist(1 1 1k)	Distance (1 1 1000)	LongMinSP(16)	Long MinSP PC (16)
Dist(1 1 10k)	Distance (1 1 10000)	LongMinSP(32)	Long MinSP PC (32)
Dist(1 1 100k)	Distance (1 1 100000)	MaxFunMinPC1	MaxFun MinPC (1)
Oracle(10 0)	GreedyOracle (10 0)	MaxFunMinPC2	MaxFun MinPC (2)
Oracle(1k 0)	GreedyOracle (1000 0)	MaxFunMinPC3	MaxFun MinPC (3)
Oracle(1k 0.5)	GreedyOracle (1000 0.5)	MinPC	MinPC
Oracle(1k 1)	GreedyOracle (1000 1)	MinSP PC	MinSP MinPC
Oracle(1k 4)	GreedyOracle (1000 4)	RDMinSP(2 1 n)	RD MinSP PC (2 1 no)
Lee	Lee	RDMinSP(4 0 n)	RD MinSP PC (4 0 no)
Long(02)	Long (02)	RDMinSP(4 1 n)	RD MinSP PC (4 1 no)
Long(04)	Long (04)	RDMinSP(8 1 y)	RD MinSP PC (8 1 yes)
Long(08)	Long (08)	RoundRobinEq	RoundRobin Eq
Long(16)	Long (16)	LongestPath1	LongestPath simple
Long(32)	Long (32)	LongestPathP	LongestPath parallel
LongIRB(04)	Long IRB (04)	PDomMaxFun	PostDom MaxFun
LongIRB(08)	Long IRB (08)	PDomMaxSP	PostDom MaxSP
LongIRB(16)	Long IRB (16)	PDomMinFun	PostDom MinFun
LongIRB(32)	Long IRB (32)	PDomMinPC	PostDom MinPC
LongIRB(64)	Long IRB (64)	PDomMinSP	PostDom MinSP
LongIRB(128)	Long IRB (128)	PDomRB	PostDom RB
LongMinPC(02)	Long MinPC (02)	PDomLong(2)	PostDom Long (2)
LongMinPC(04)	Long MinPC (04)	PDomLong(4)	PostDom Long (4)
LongMinPC(08)	Long MinPC (08)	PDomLong(8)	PostDom Long (8)
LongMinPC(16)	Long MinPC (16)	PDomLong(16)	PostDom Long (16)
LongMinPC(32)	Long MinPC (32)	PDomLong(32)	PostDom Long (32)
LongMinSP(02)	Long MinSP PC (02)		

Table 5.2. Subtitles of heuristics

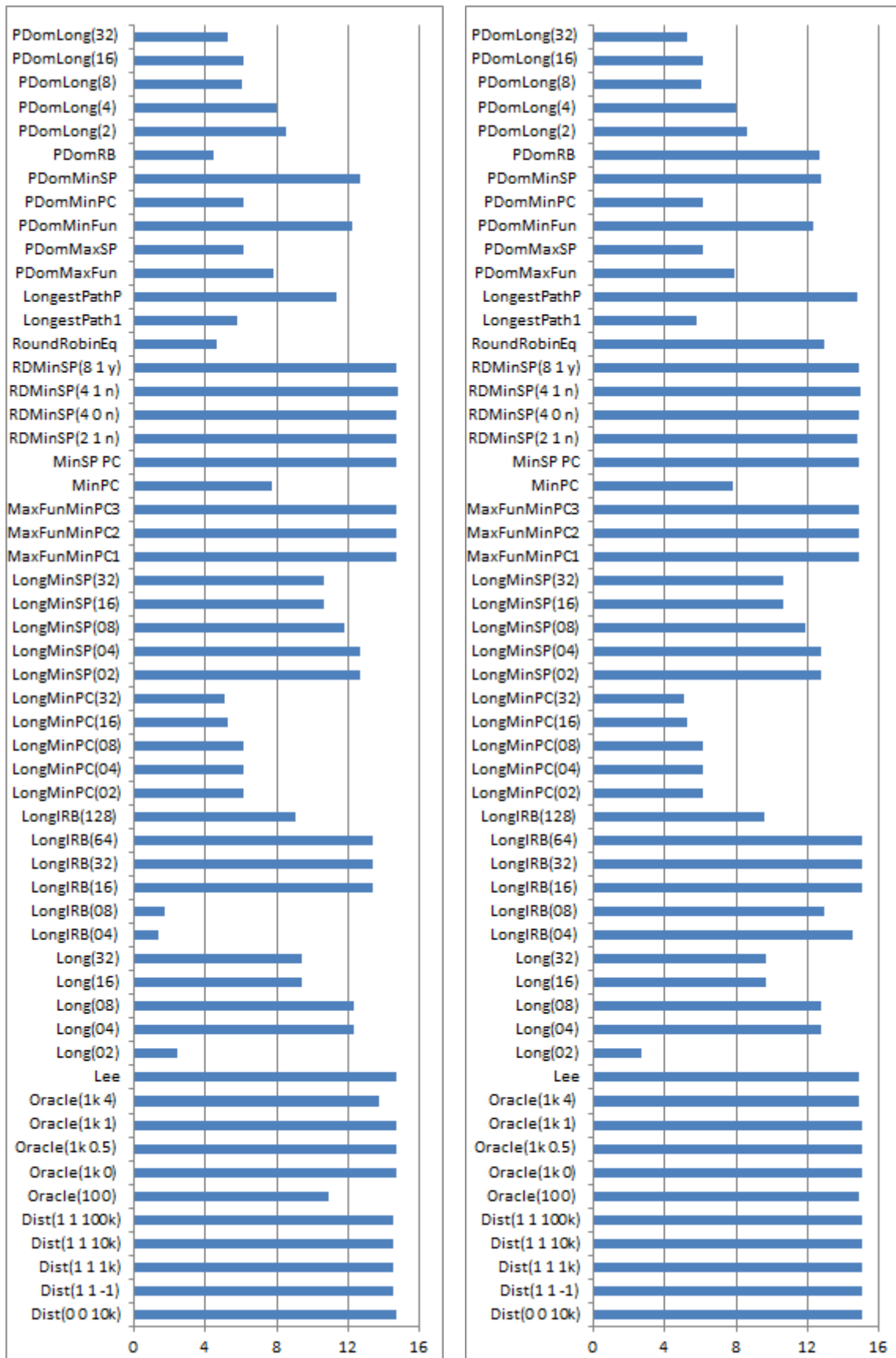


Figure 5.1. Heuristics with blackscholes

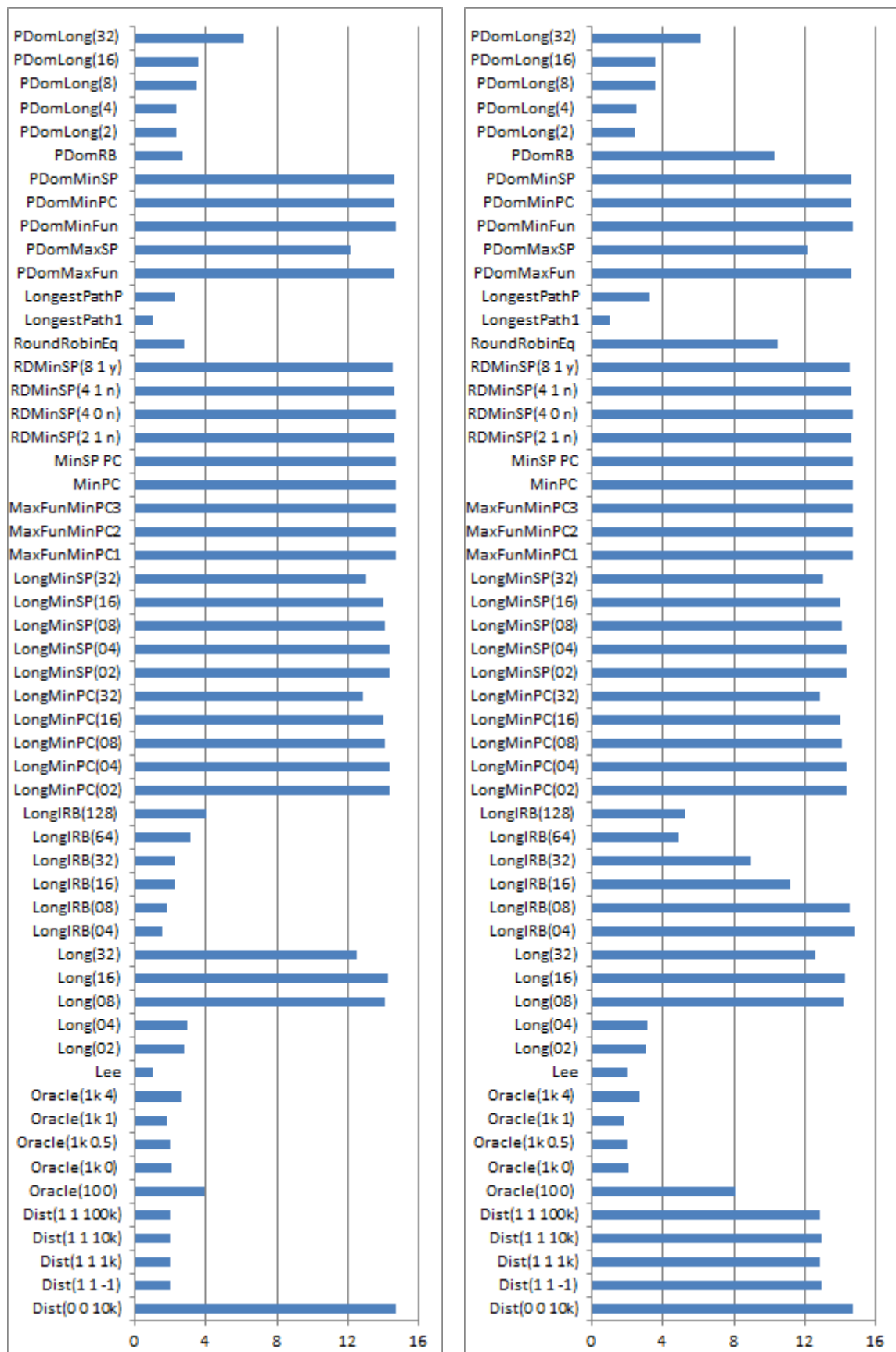


Figure 5.2. Heuristics with bodytrack

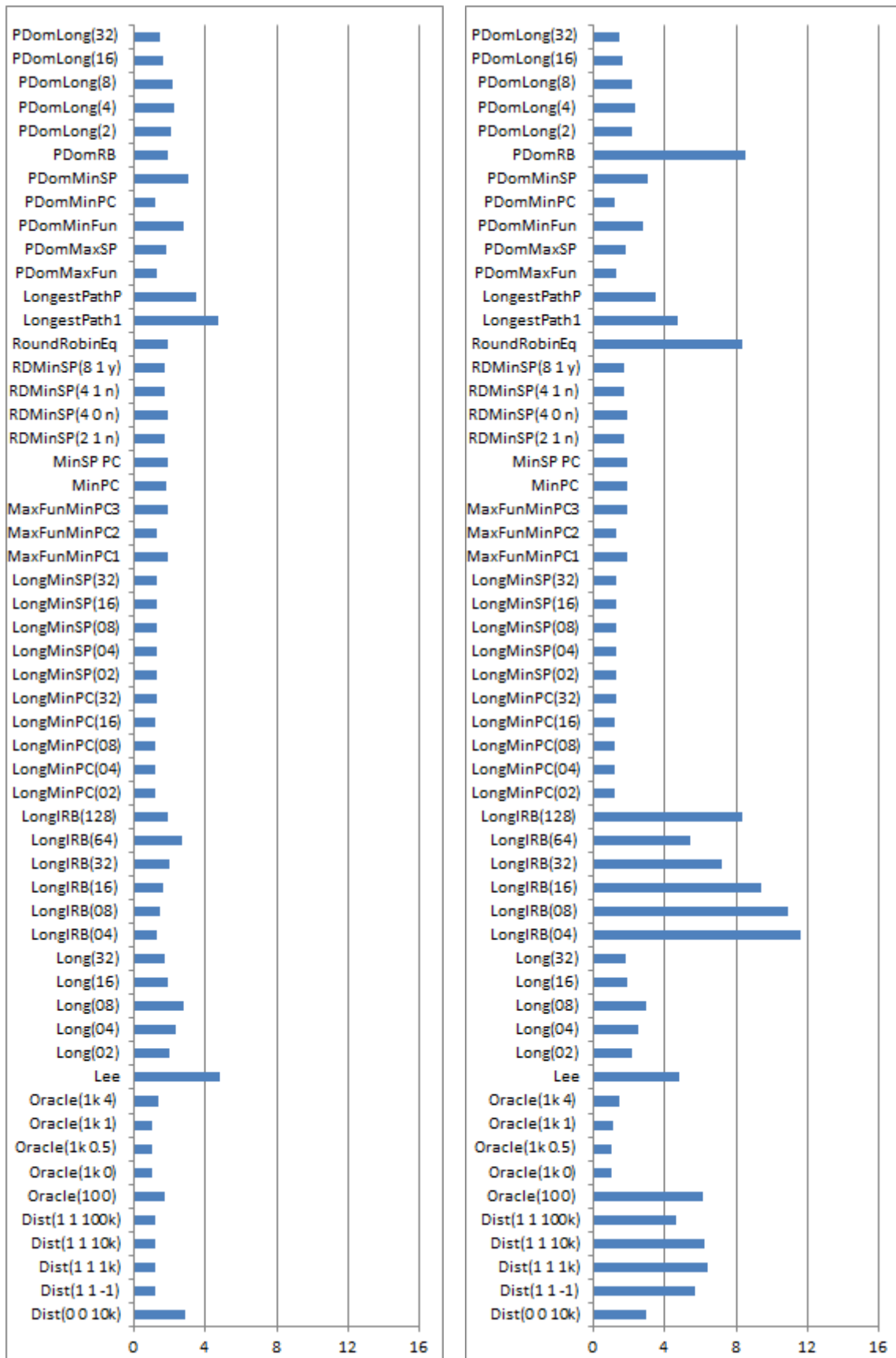


Figure 5.3. Heuristics with fluidanimate



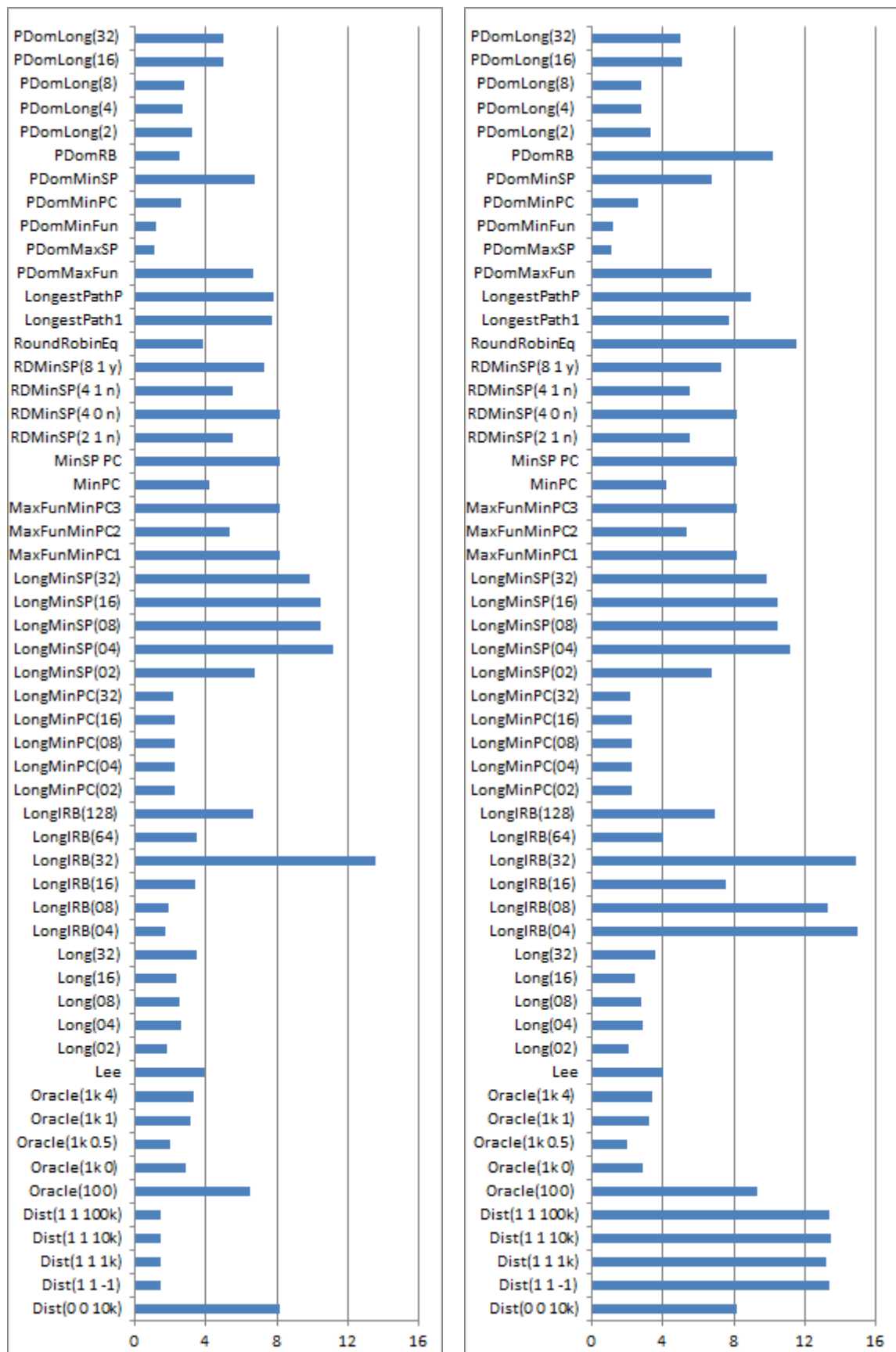


Figure 5.4. Heuristics with swaptions

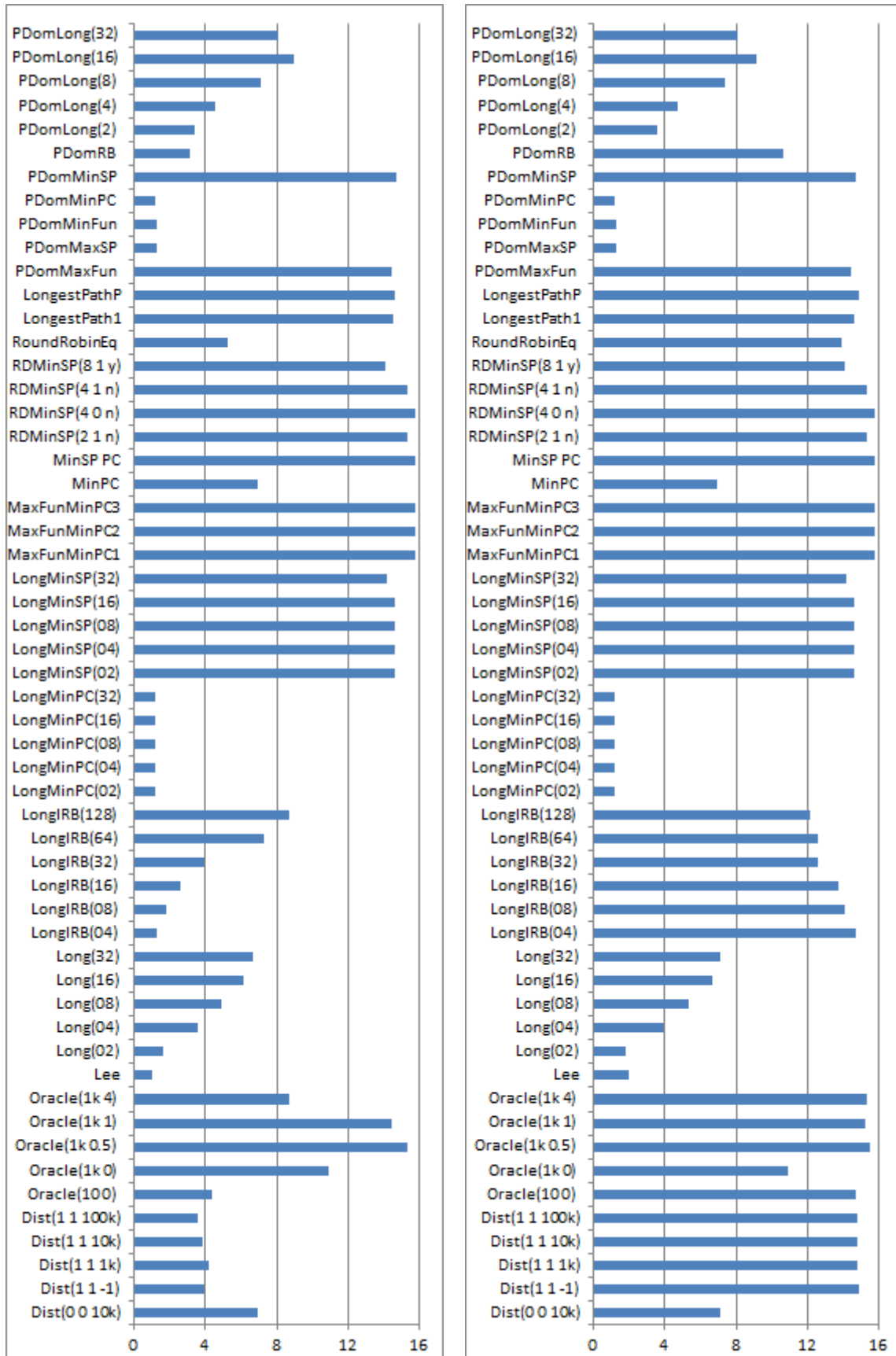


Figure 5.5. Heuristics with tachyon

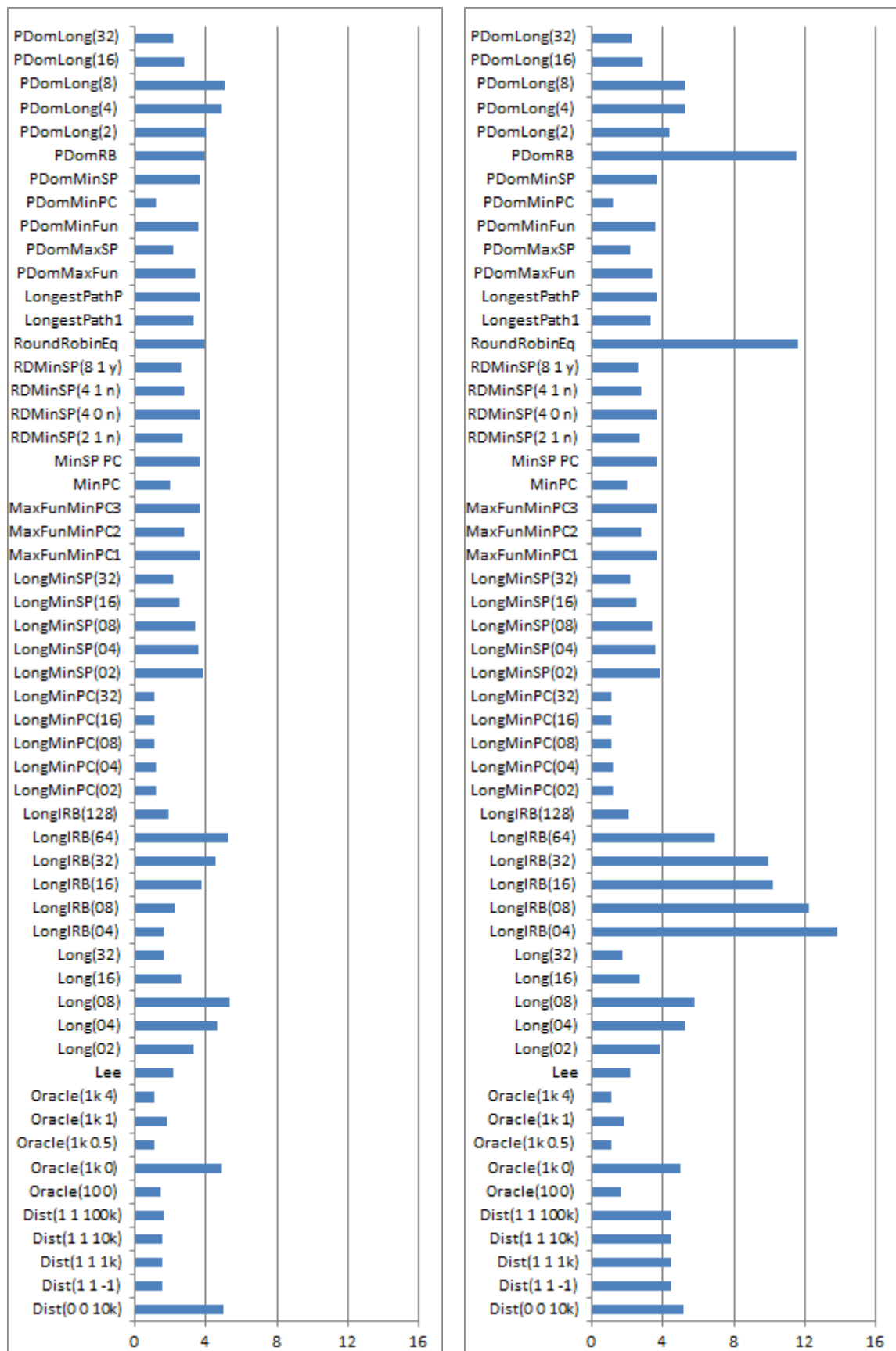


Figure 5.6. Heuristics with barnes

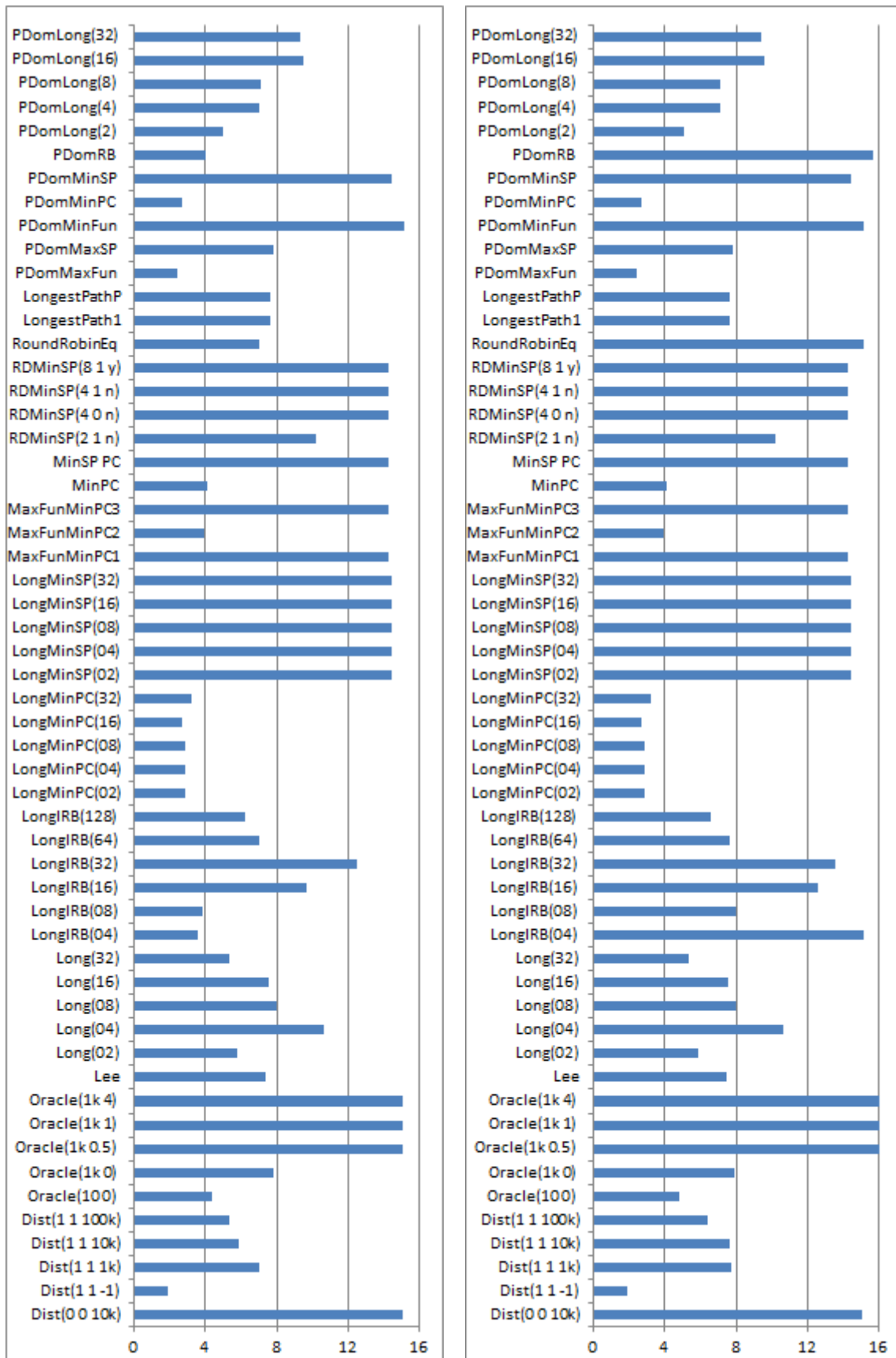


Figure 5.7. Heuristics with fft

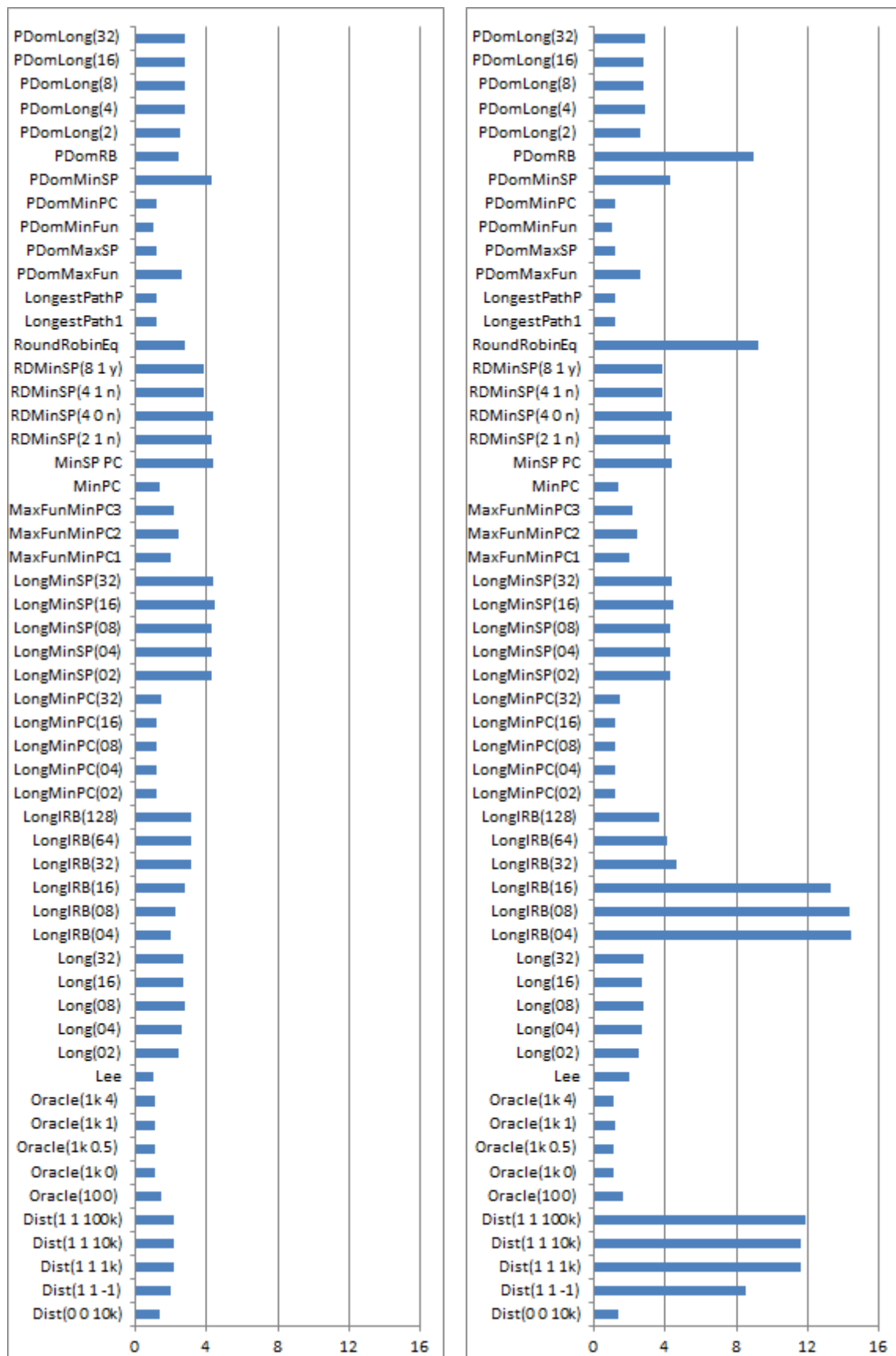


Figure 5.8. Heuristics with fmm

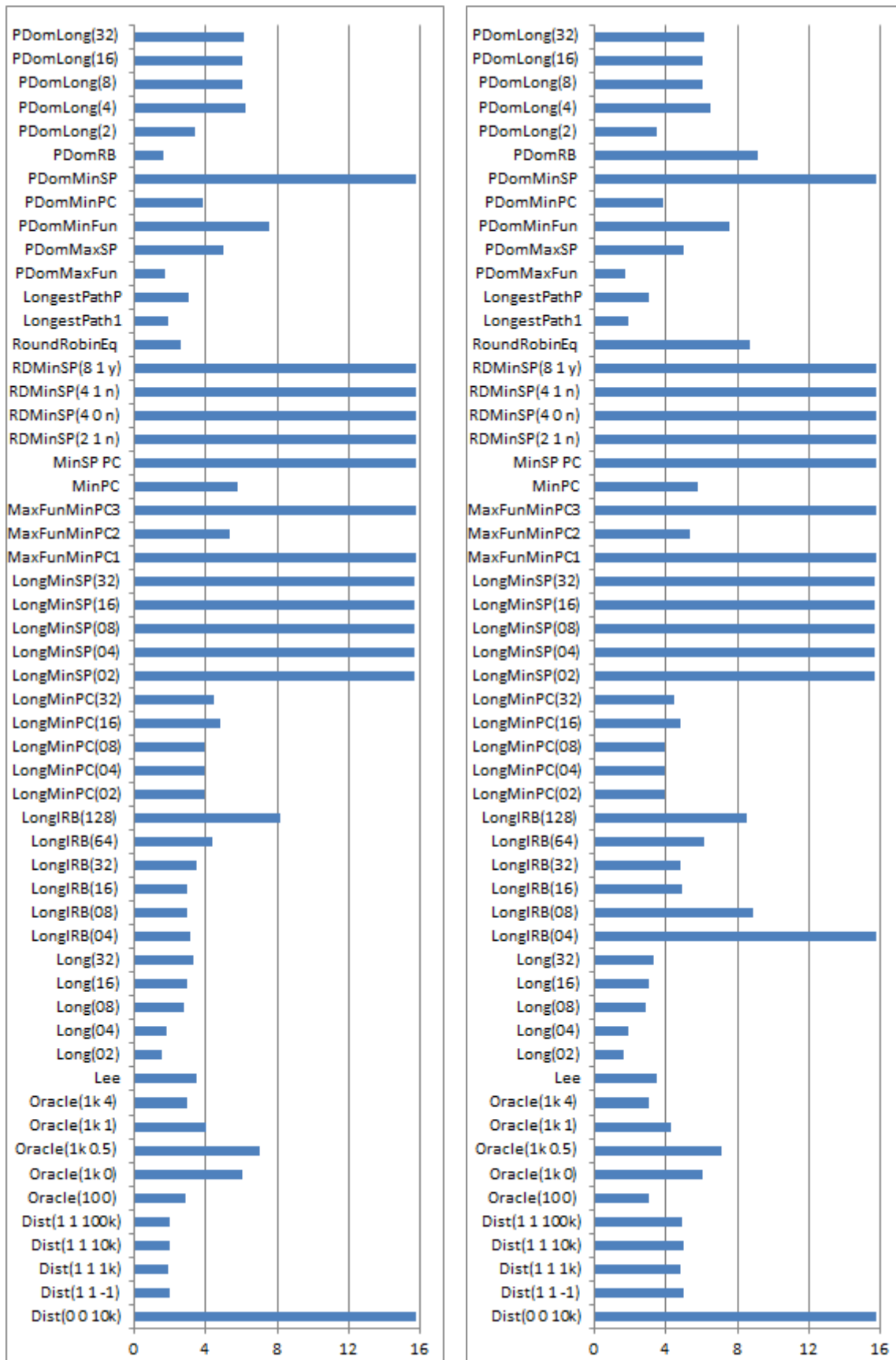


Figure 5.9. Heuristics with ocean\_ncp

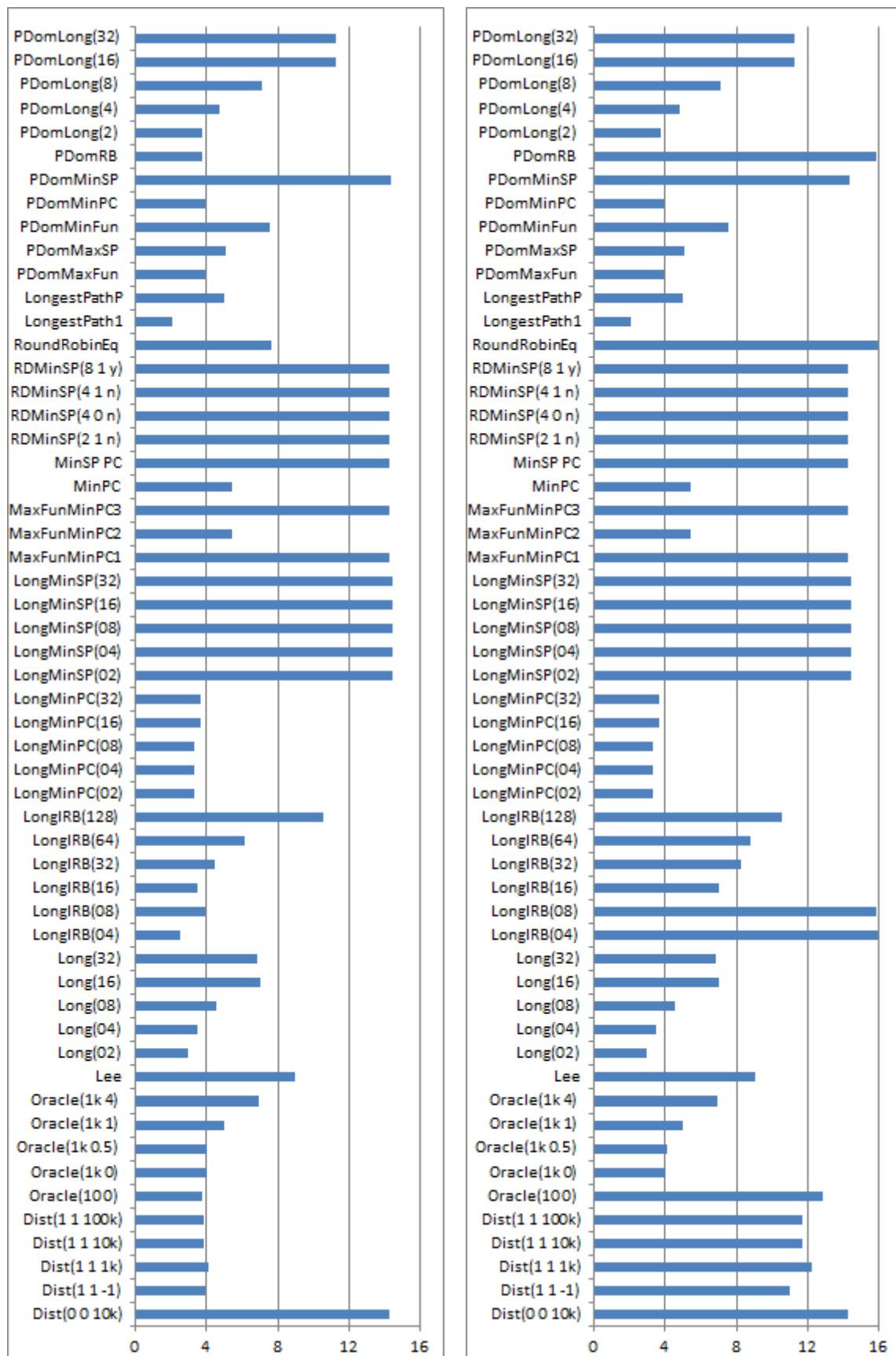


Figure 5.10. Heuristics with radix

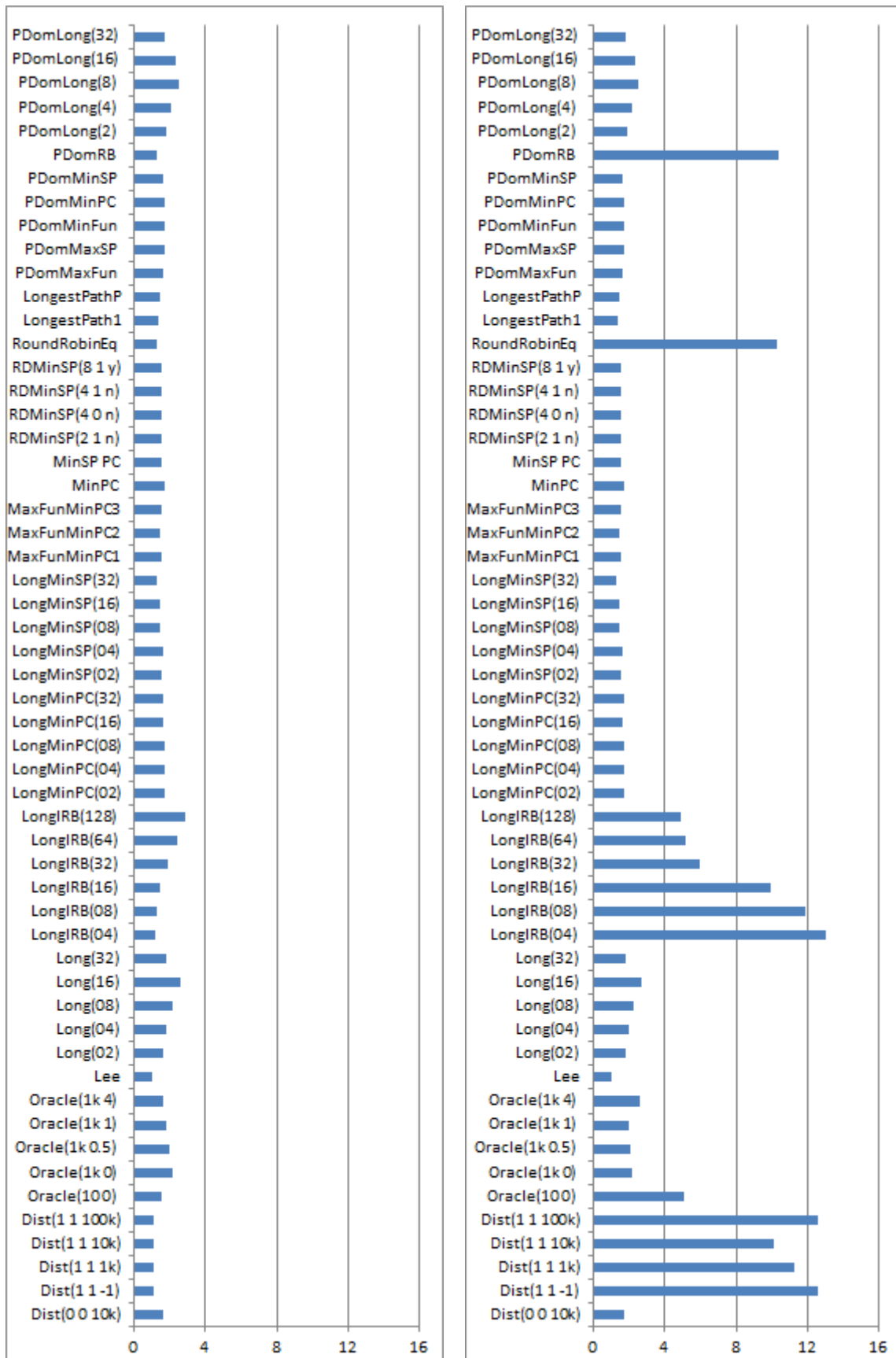


Figure 5.11. Heuristics with volrend



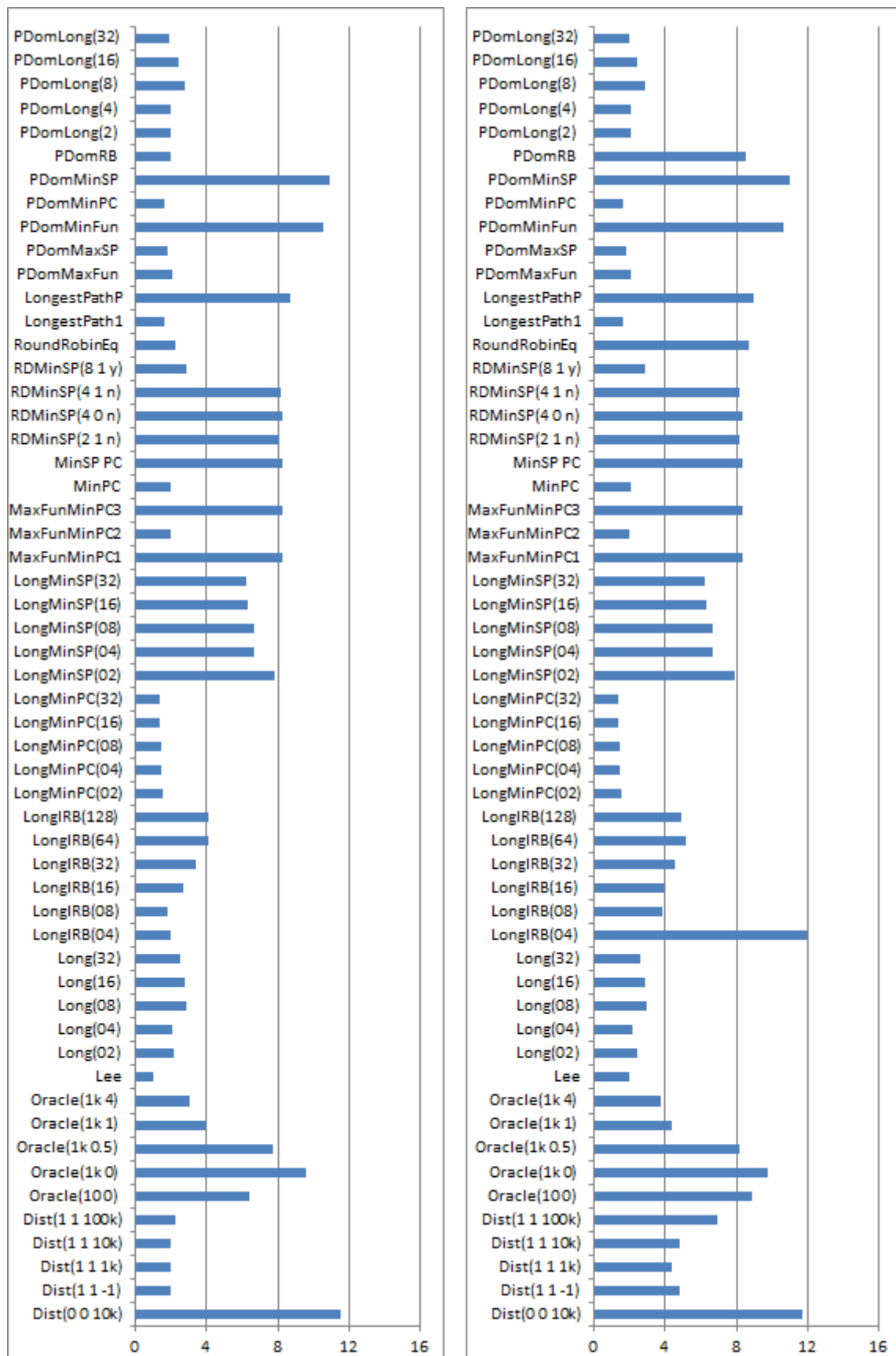


Figure 5.12. Heuristics with water\_nsquared

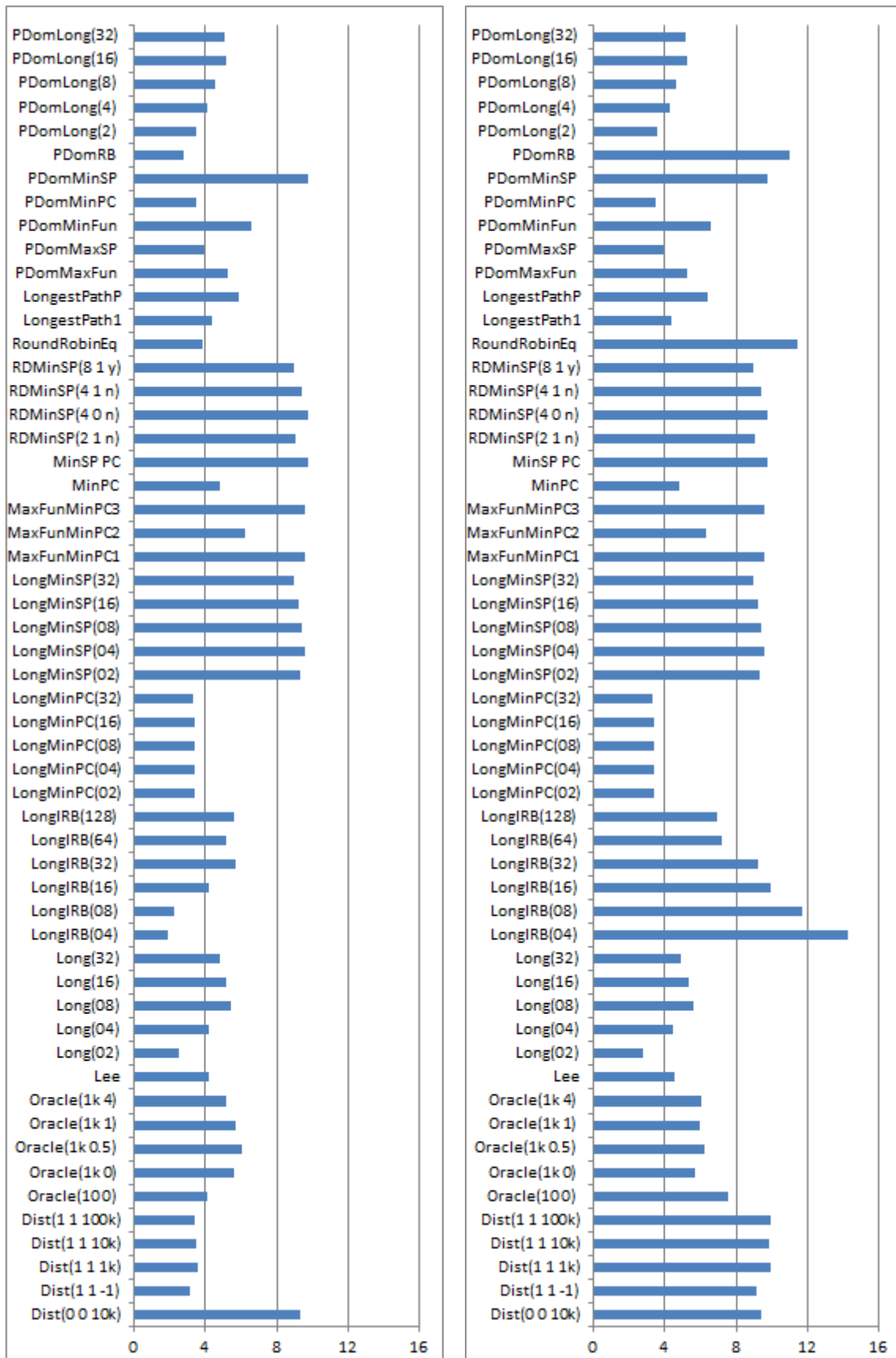


Figure 5.13. Average result of heuristics

Heuristic	barn	blacks	body	fft	fluid	fmm	ocean	radix	swapt	tach	volr	water	avg
Dist(0 0 10k)	5,01	14,73	14,69	15,09	2,83	1,38	15,78	14,25	8,14	6,95	1,66	11,53	9,34
Dist(1 1 -1)	1,57	14,52	1,95	1,89	1,17	2,01	1,97	3,96	1,42	3,89	1,09	2,00	3,12
Dist(1 1 1k)	1,56	14,52	1,95	7,03	1,17	2,13	1,92	4,11	1,41	4,17	1,09	1,93	3,58
Dist(1 1 10k)	1,56	14,52	1,93	5,89	1,17	2,13	1,99	3,85	1,45	3,81	1,09	1,98	3,45
Dist(1 1 100k)	1,57	14,52	1,97	5,33	1,17	2,13	1,96	3,85	1,43	3,60	1,09	2,24	3,41
Oracle(10 0)	1,42	10,92	3,91	4,36	1,67	1,40	2,86	3,74	6,50	4,35	1,49	6,44	4,09
Oracle(1k 0)	4,89	14,73	2,04	7,78	1,02	1,07	6,02	3,98	2,83	10,88	2,17	9,61	5,58
Oracle(1k 0.5)	1,08	14,73	1,94	15,09	1,02	1,07	6,98	4,05	1,98	15,33	1,97	7,76	6,08
Oracle(1k 1)	1,79	14,73	1,79	15,09	1,04	1,12	4,01	4,99	3,15	14,44	1,84	3,93	5,66
Oracle(1k 4)	1,06	13,75	2,62	15,09	1,36	1,09	2,92	6,92	3,31	8,73	1,64	3,06	5,13
Lee	2,15	14,73	1,01	7,40	4,81	1,01	3,50	9,00	3,96	1,01	1,02	1,01	4,22
Long(02)	3,32	2,44	2,81	5,78	1,94	2,38	1,57	2,98	1,77	1,63	1,66	2,15	2,53
Long(04)	4,66	12,36	2,91	10,60	2,32	2,62	1,82	3,48	2,58	3,52	1,83	2,02	4,23
Long(08)	5,32	12,36	14,11	7,95	2,76	2,74	2,77	4,57	2,54	4,85	2,13	2,83	5,41
Long(16)	2,57	9,44	14,30	7,54	1,86	2,64	2,97	7,00	2,35	6,11	2,56	2,80	5,18
Long(32)	1,66	9,44	12,53	5,30	1,72	2,69	3,29	6,82	3,52	6,68	1,80	2,52	4,83
LongIRB(04)	1,50	5,23	11,98	4,55	1,43	2,71	2,84	6,66	6,45	6,23	1,43	1,80	4,40
LongIRB(08)	1,63	1,34	1,53	3,53	1,23	1,97	3,13	2,48	1,71	1,27	1,17	1,97	1,91
LongIRB(16)	3,75	13,37	2,22	9,69	1,58	2,81	2,91	3,46	3,41	2,56	1,48	2,70	4,16
LongIRB(32)	4,56	13,37	2,27	12,51	2,01	3,16	3,45	4,48	13,53	3,88	1,89	3,37	5,71
LongIRB(64)	5,26	13,37	3,12	7,04	2,68	3,10	4,36	6,10	3,49	7,31	2,39	4,09	5,19
LongIRB(128)	2,26	1,74	1,75	3,85	1,40	2,25	2,92	3,94	1,89	1,81	1,28	1,82	2,24
LongMinPC(02)	1,18	6,10	14,37	2,87	1,17	1,16	3,88	3,29	2,28	1,16	1,69	1,52	3,39
LongMinPC(04)	1,18	6,10	14,37	2,87	1,18	1,16	3,88	3,30	2,28	1,16	1,70	1,47	3,39
LongMinPC(08)	1,07	6,10	14,08	2,87	1,18	1,16	3,88	3,30	2,28	1,16	1,70	1,47	3,35
LongMinPC(16)	1,07	5,27	13,99	2,65	1,21	1,17	4,78	3,66	2,21	1,16	1,63	1,36	3,35
LongMinPC(32)	1,08	5,05	12,83	3,18	1,25	1,43	4,41	3,66	2,18	1,16	1,66	1,36	3,27
LongMinSP(02)	3,81	12,68	14,37	14,48	1,27	4,24	15,72	14,46	6,73	14,64	1,56	7,83	9,32
LongMinSP(04)	3,54	12,68	14,37	14,48	1,27	4,23	15,72	14,46	11,14	14,64	1,59	6,64	9,56
LongMinSP(08)	3,39	11,80	14,08	14,48	1,27	4,23	15,70	14,46	10,50	14,64	1,47	6,64	9,39
LongMinSP(16)	2,46	10,63	13,99	14,48	1,27	4,42	15,70	14,46	10,50	14,64	1,47	6,29	9,19
LongMinSP(32)	2,11	10,63	13,06	14,48	1,30	4,40	15,70	14,46	9,88	14,15	1,29	6,21	8,97
MaxFunMinPC1	3,66	14,73	14,69	14,27	1,90	2,02	15,78	14,25	8,14	15,74	1,57	8,28	9,59
MaxFunMinPC2	2,79	14,73	14,69	3,94	1,30	2,40	5,34	5,45	5,35	15,74	1,40	1,99	6,26
MaxFunMinPC3	3,66	14,73	14,69	14,27	1,90	2,14	15,78	14,25	8,14	15,74	1,57	8,28	9,60
MinPC	2,00	7,73	14,69	4,09	1,82	1,39	5,78	5,45	4,14	6,89	1,66	1,99	4,80
MinSP_MinPC	3,66	14,73	14,69	14,27	1,90	4,35	15,78	14,25	8,14	15,74	1,57	8,28	9,78
RDMinSP(2 1 n)	2,69	14,69	14,66	10,20	1,75	4,25	15,78	14,25	5,47	15,33	1,54	8,11	9,06
RDMinSP(4 0 n)	3,66	14,73	14,69	14,27	1,90	4,35	15,78	14,25	8,14	15,74	1,57	8,28	9,78
RDMinSP(4 1 n)	2,77	14,84	14,65	14,27	1,75	3,85	15,78	14,25	5,47	15,33	1,54	8,12	9,39
RDMinSP(8 1 y)	2,57	14,73	14,56	14,27	1,67	3,80	15,78	14,25	7,24	14,10	1,56	2,86	8,95
RoundRobinEq	3,95	4,64	2,74	7,02	1,93	2,74	2,60	7,63	3,82	5,27	1,29	2,26	3,82
LongestPath1	3,28	5,76	1,04	7,60	4,69	1,17	1,85	2,03	7,69	14,58	1,32	1,64	4,39
LongestPathP	3,66	11,38	2,28	7,63	3,50	1,17	3,00	4,96	7,79	14,63	1,43	8,71	5,84
PDomMaxFun	3,42	7,85	14,60	2,44	1,29	2,57	1,67	3,89	6,70	14,43	1,63	2,02	5,21
PDomMaxSP	2,17	6,10	12,17	7,80	1,79	1,15	5,01	5,04	1,08	1,26	1,73	1,75	3,92
PDomMinFun	3,55	12,25	14,72	15,16	2,76	1,00	7,56	7,52	1,13	1,26	1,73	10,52	6,60
PDomMinPC	1,17	6,10	14,60	2,65	1,16	1,19	3,79	3,89	2,59	1,16	1,71	1,59	3,47
PDomMinSP	3,63	12,66	14,60	14,42	3,01	4,29	15,75	14,40	6,77	14,70	1,63	10,87	9,73
PDomRB	3,88	4,48	2,65	4,00	1,85	2,44	1,64	3,71	2,51	3,14	1,26	2,01	2,80
PDomLong(2)	4,00	8,52	2,30	4,98	2,10	2,51	3,41	3,74	3,22	3,40	1,77	1,99	3,49
PDomLong(4)	4,87	7,96	2,35	6,99	2,25	2,79	6,24	4,76	2,69	4,51	2,09	2,00	4,12
PDomLong(8)	5,04	6,04	3,52	7,09	2,11	2,72	6,06	7,10	2,77	7,14	2,46	2,81	4,57
PDomLong(16)	2,79	6,16	3,54	9,53	1,64	2,74	6,01	11,25	5,02	8,93	2,28	2,42	5,19
PDomLong(32)	2,19	5,29	6,11	9,35	1,45	2,79	6,13	11,25	4,98	8,07	1,74	1,92	5,11

Table 5.3. DLP of all benchmarks with all heuristics

Heuristic	barn	blacks	body	fft	fluid	fmm	ocean	radix	swapt	tach	volr	water	avg
Dist(0 0 10k)	5,12	15,07	14,69	15,09	2,92	1,38	15,79	14,26	8,17	7,09	1,67	11,75	9,42
Dist(1 1 -1)	4,45	15,07	12,93	1,90	5,66	8,53	4,95	11,03	13,35	14,88	12,62	4,81	9,18
Dist(1 1 1k)	4,46	15,07	12,89	7,75	6,43	11,59	4,81	12,21	13,22	14,84	11,27	4,37	9,91
Dist(1 1 10k)	4,45	15,07	12,91	7,65	6,24	11,63	4,97	11,72	13,45	14,83	10,16	4,85	9,83
Dist(1 1 100k)	4,45	15,07	12,86	6,35	4,60	11,85	4,91	11,72	13,42	14,79	12,62	6,90	9,96
Oracle(10 0)	1,64	14,87	8,04	4,79	6,13	1,65	3,03	12,87	9,33	14,68	5,03	8,89	7,58
Oracle(1k 0)	4,94	15,04	2,06	7,88	1,02	1,09	6,08	3,98	2,83	10,92	2,17	9,75	5,65
Oracle(1k 0.5)	1,08	15,06	1,96	16,00	1,02	1,09	7,13	4,05	1,99	15,54	2,04	8,18	6,26
Oracle(1k 1)	1,81	15,07	1,81	16,00	1,04	1,13	4,26	4,99	3,17	15,27	2,00	4,38	5,91
Oracle(1k 4)	1,07	14,93	2,66	16,00	1,40	1,12	3,02	6,93	3,39	15,32	2,60	3,74	6,01
Lee	2,16	14,85	1,96	7,45	4,84	1,96	3,50	9,02	3,97	1,96	1,02	1,96	4,55
Long(02)	3,81	2,68	3,04	5,83	2,14	2,51	1,64	2,98	2,07	1,84	1,81	2,39	2,73
Long(04)	5,26	12,74	3,14	10,67	2,54	2,72	1,89	3,49	2,88	3,95	1,97	2,16	4,45
Long(08)	5,74	12,74	14,16	8,01	2,96	2,81	2,86	4,58	2,74	5,34	2,26	2,99	5,60
Long(16)	2,65	9,67	14,32	7,58	1,90	2,70	3,06	7,00	2,43	6,66	2,66	2,89	5,29
Long(32)	1,70	9,67	12,62	5,33	1,75	2,73	3,34	6,82	3,58	7,10	1,83	2,58	4,92
LongIRB(04)	13,83	14,50	14,77	15,19	11,65	14,41	15,74	15,98	14,95	14,68	13,03	12,01	14,23
LongIRB(08)	12,21	12,92	14,53	7,99	10,93	14,32	8,90	15,83	13,34	14,14	11,85	3,85	11,73
LongIRB(16)	9,94	15,07	8,96	13,56	7,17	4,64	4,84	8,27	14,87	12,57	5,93	4,55	9,20
LongIRB(32)	6,96	15,07	4,93	7,65	5,43	4,11	6,14	8,83	4,03	12,56	5,14	5,19	7,17
LongIRB(64)	2,06	9,61	5,29	6,54	8,37	3,69	8,52	10,52	6,95	12,12	4,92	4,92	6,96
LongIRB(128)	10,23	15,07	11,20	12,59	9,43	13,32	4,87	7,03	7,53	13,70	9,96	3,88	9,90
LongMinPC(02)	1,18	6,11	14,38	2,87	1,18	1,16	3,88	3,29	2,28	1,16	1,70	1,52	3,39
LongMinPC(04)	1,18	6,11	14,38	2,87	1,18	1,16	3,88	3,30	2,28	1,16	1,70	1,47	3,39
LongMinPC(08)	1,07	6,11	14,08	2,87	1,18	1,16	3,88	3,30	2,28	1,16	1,70	1,47	3,36
LongMinPC(16)	1,07	5,28	14,00	2,65	1,21	1,17	4,78	3,66	2,21	1,16	1,63	1,36	3,35
LongMinPC(32)	1,08	5,06	12,84	3,18	1,25	1,43	4,41	3,66	2,18	1,16	1,66	1,36	3,27
LongMinSP(02)	3,82	12,74	14,38	14,48	1,27	4,24	15,72	14,46	6,74	14,66	1,56	7,87	9,33
LongMinSP(04)	3,55	12,74	14,38	14,48	1,27	4,23	15,72	14,46	11,16	14,66	1,59	6,68	9,58
LongMinSP(08)	3,40	11,88	14,09	14,48	1,27	4,23	15,71	14,46	10,51	14,66	1,47	6,68	9,40
LongMinSP(16)	2,47	10,67	14,00	14,48	1,27	4,42	15,71	14,46	10,51	14,66	1,47	6,31	9,20
LongMinSP(32)	2,12	10,67	13,07	14,48	1,30	4,40	15,70	14,46	9,88	14,16	1,29	6,24	8,98
MaxFunMinPC1	3,67	14,85	14,69	14,27	1,90	2,02	15,79	14,25	8,16	15,74	1,57	8,34	9,60
MaxFunMinPC2	2,79	14,85	14,69	3,94	1,30	2,40	5,34	5,45	5,37	15,74	1,40	1,99	6,27
MaxFunMinPC3	3,67	14,85	14,69	14,27	1,90	2,14	15,79	14,25	8,16	15,74	1,57	8,34	9,61
MinPC	2,02	7,79	14,69	4,09	1,85	1,40	5,79	5,45	4,15	6,91	1,66	2,02	4,82
MinSP_PC	3,67	14,85	14,69	14,27	1,90	4,35	15,79	14,25	8,16	15,74	1,57	8,34	9,80
RDMinSP(2 1 n)	2,70	14,81	14,66	10,20	1,75	4,26	15,79	14,25	5,48	15,33	1,54	8,17	9,08
RDMinSP(4 0 n)	3,67	14,85	14,69	14,27	1,90	4,35	15,79	14,25	8,16	15,74	1,57	8,34	9,80
RDMinSP(4 1 n)	2,78	14,96	14,65	14,27	1,75	3,85	15,79	14,25	5,48	15,33	1,54	8,18	9,40
RDMinSP(8 1 y)	2,58	14,85	14,56	14,27	1,67	3,80	15,79	14,25	7,26	14,10	1,56	2,88	8,96
RoundRobinEq	11,57	12,97	10,43	15,16	8,37	9,26	8,70	15,93	11,57	13,88	10,25	8,69	11,40
LongestPath1	3,28	5,78	1,04	7,60	4,69	1,17	1,85	2,03	7,70	14,59	1,32	1,64	4,39
LongestPathP	3,69	14,85	3,17	7,63	3,51	1,17	3,02	4,96	8,93	14,91	1,44	8,97	6,35
PDomMaxFun	3,43	7,89	14,60	2,44	1,29	2,57	1,67	3,89	6,72	14,44	1,63	2,03	5,22
PDomMaxSP	2,19	6,11	12,17	7,80	1,80	1,15	5,01	5,04	1,08	1,26	1,73	1,76	3,93
PDomMinFun	3,57	12,31	14,72	15,16	2,80	1,00	7,57	7,52	1,13	1,26	1,74	10,62	6,62
PDomMinPC	1,17	6,11	14,60	2,65	1,16	1,19	3,79	3,89	2,59	1,16	1,71	1,59	3,47
PDomMinSP	3,64	12,74	14,60	14,42	3,01	4,29	15,76	14,40	6,78	14,72	1,63	10,97	9,75
PDomRB	11,54	12,67	10,28	15,65	8,48	8,98	9,10	15,84	10,19	10,63	10,39	8,52	11,02
PDomLong(2)	4,37	8,59	2,44	5,07	2,19	2,57	3,48	3,74	3,32	3,58	1,87	2,07	3,61
PDomLong(4)	5,20	8,01	2,48	7,08	2,33	2,84	6,45	4,76	2,74	4,69	2,18	2,07	4,24
PDomLong(8)	5,26	6,05	3,59	7,14	2,14	2,77	6,08	7,10	2,81	7,34	2,55	2,86	4,64
PDomLong(16)	2,83	6,17	3,60	9,57	1,66	2,78	6,04	11,25	5,05	9,13	2,32	2,45	5,24
PDomLong(32)	2,23	5,29	6,15	9,38	1,47	2,82	6,15	11,25	5,00	8,12	1,76	1,94	5,13

Table 5.4. Throughput of all benchmarks with all heuristics

### 5.1.1 Discussion

Despite the applications being data-parallel, many of them are difficult to synchronize. Consequently, no heuristic is good with all of them. This diversity leads to curious situations: sometimes a heuristic that is bad in most cases is good for a specific benchmark. This case is illustrated by Lee's heuristic, which is good with `fluidanimate`.

In the end, we have not been able to identify a heuristic that outperform all the others. We are inclined to think that the hardware industry needs an adaptive heuristic that learns the characteristics of the benchmarks and is able to capitalize in these specific features.

Heuristics that are good in normal SIMD will not have worse performance in opportunistic SIMD. Because the performance in opportunistic SIMD is identified by the instruction throughput and instruction throughput includes DLP, as explained in section 1.5.2.

Heuristics that provide higher TLP, such as some variations of Distance, Greedy Oracle and Long-IRB, have better instruction throughput than the best heuristics for normal SIMD, which have high DLP. These heuristics were created to have high TLP instead of high DLP and, as consequence, they have better performance in an opportunistic SIMD architecture. We have been able to observe this behavior in benchmarks that are difficult to synchronize, such as `fluidanimate`, `barnes`, `fmm` and `volrend`. Therefore, if we do not have a heuristic with good performance for normal SIMD, we have a heuristic with better performance in opportunistic SIMD.

MaxFun-MinPC has almost the same result as MinSP-MinPC. It was expected because they follow the same principle of regarding the activation records in the call stack and using the smaller PC. MinSP-MinPC has good results with benchmarks that are easy to synchronize. These benchmarks have functions that Min-PC can synchronize easily. Min-PC is improved by MaxFun-MinPC and MinSP-MinPC because it handles the effects of different functions calls as explained in chapter 3.

We can see that Distance and MinSP-MinPC are good heuristics in most cases, but they have different results. Distance with `tachyon` has bad DLP, but good TLP. The instruction throughput is high and therefore it is good in opportunistic SIMD. In common SIMD, while Distance is better with `barnes` and bad with `tachyon`, which is easy to synchronize, MinSP-MinPC is good with `tachyon` and bad with `barnes`, which is difficult to synchronize.

The heuristics that use the longest path, which is used by the reference implementation of thread frontiers, did not have good results. We speculate that the reason is a bad algorithm to find good approximated longest path, because many

functions are large. We do not have a depth study that measures how good is this policy.

Long’s heuristics have many variations, which may be good in different cases. As a consequence, we could not find a generic version of Long’s heuristic that is better generally. We observed that tables with sizes larger than 32 are bad and some very small sizes are for rare specific cases. The size of the table cannot be large or small. Small table does not have enough space to keep the distance of divergent threads. But a large table causes lots of false positives, because a thread that is ahead may find its PC in the table of a thread that is behind due to loops.

The Round-Robin heuristic simulates the execution in a MIMD machine, therefore it has the lowest DLP, which is 1, and highest TLP, which should be the number of threads, but it is a little smaller due to the small unbalance in the distribution of the thread load. Round-Robin-Eq is a heuristic similar to Round-Robin, but it shares instruction fetches when it is possible. These heuristics illustrate the extreme cases with high TLP.

Finally, we can define an optimal subset of heuristics. We exclude heuristics that cannot be implemented in hardware or are very expensive, such as Long, Distance and GreedyOracle. The subset would include MinSP-MinPC, MaxFun-MinPC and PostDom with MinSP. Nevertheless LongestPath heuristics need further improvements in the implementation and studies in the behavior when we have multiple function calls.

## 5.2 Execute-identical instructions

Here we present histograms that show how often we have a specific number, from 1 to 16, of active threads. In other words, the histograms show in how many cycles we have N threads in execution when an instruction is being executed. This information lets us know how many threads are divergent on average. Consequently, it gives us the average number of waiting threads.

The instructions of each bar of the histogram are fetch-identical. Among them, the charts also show how many instructions are execute-identical. As explained in section 1.2, execute-identical instructions may be existing only if the instructions are fetch-identical.

In these charts, the X axis denotes the number of threads in execution and the Y axis denotes the number of cycles in millions that it happens. The execute-identical instructions are in black, while the non-execute-identical instructions are in gray.

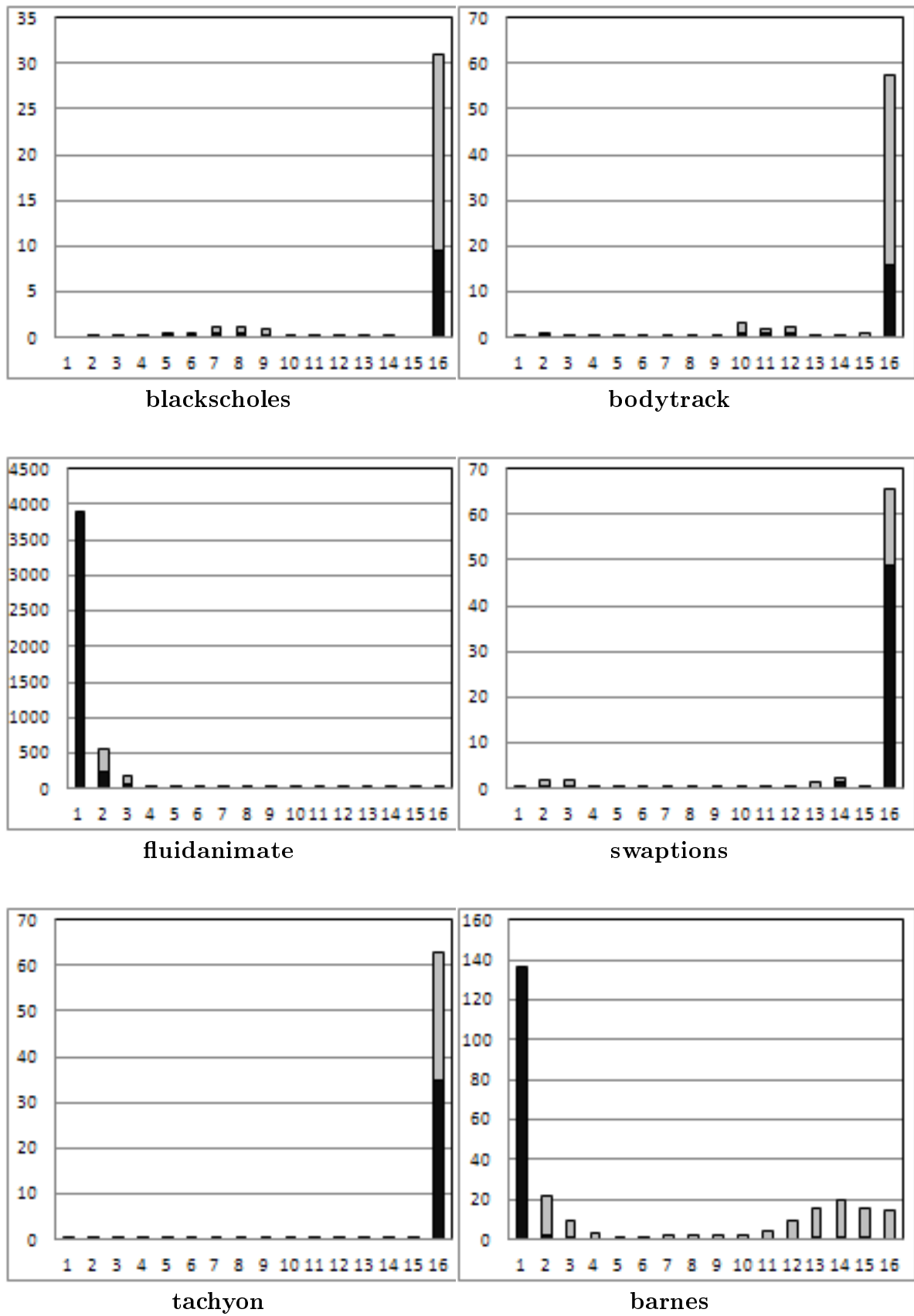


Figure 5.14. Histogram (1/2)

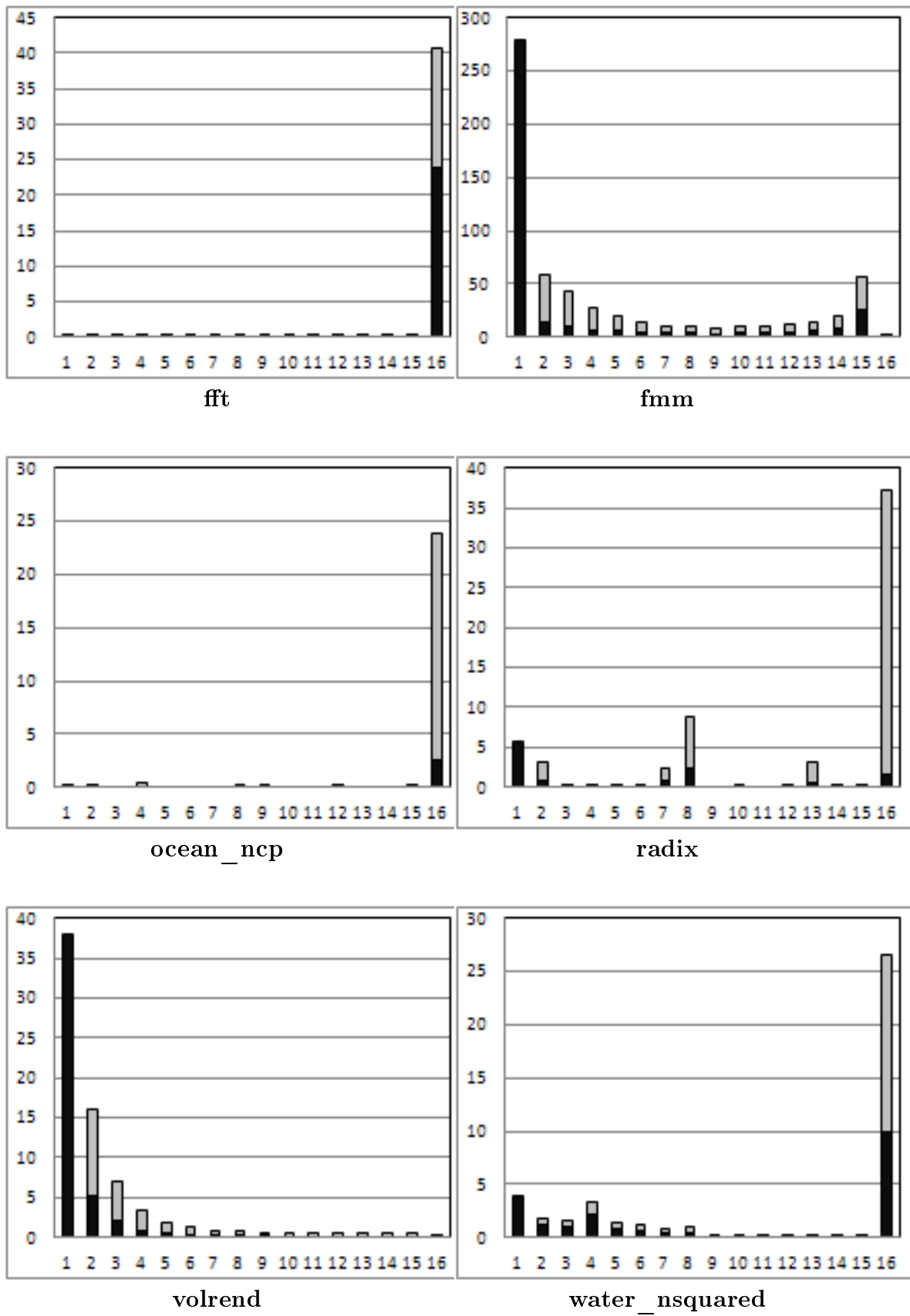


Figure 5.15. Histogram (2/2)



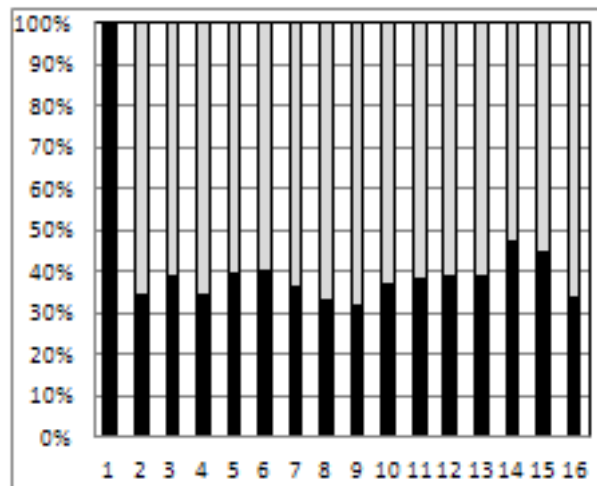


Figure 5.16. Average of execute-identical instructions.

### 5.2.1 Discussion

As presented in section 5.1, there is no heuristic that is good in all cases, but there are heuristics that are generally bad, but good with a specific benchmark. When a heuristic that is able to keep the threads of some benchmarks synchronized, we have all threads active during almost all the time. The benchmarks that we could not synchronize well have their execution almost serial with single active thread during most times. These benchmarks are fluidanimate, barnes, fmm and volrend.

We speculate that most of the histograms concentrate on one or sixteen because we have used the best heuristics for each benchmark, when doing this analysis. The benchmarks that are difficult to synchronize have their execution almost serial with any heuristic.

We can see that the number of execute-identical instructions is high. The average of execute-identical instructions of all benchmarks when we have 16 executing active is 33.670%. And on the average we had 37.697% threads active at any time.

## 5.3 Memory access pattern

The following figures show how often each memory access pattern is found in our simulation using those data-parallel applications. These patterns are described in section 1.5.4. In these charts, white denotes scattered memory accesses, gray means affine memory accesses and black means uniform memory accesses. The X axis is the number of active threads.

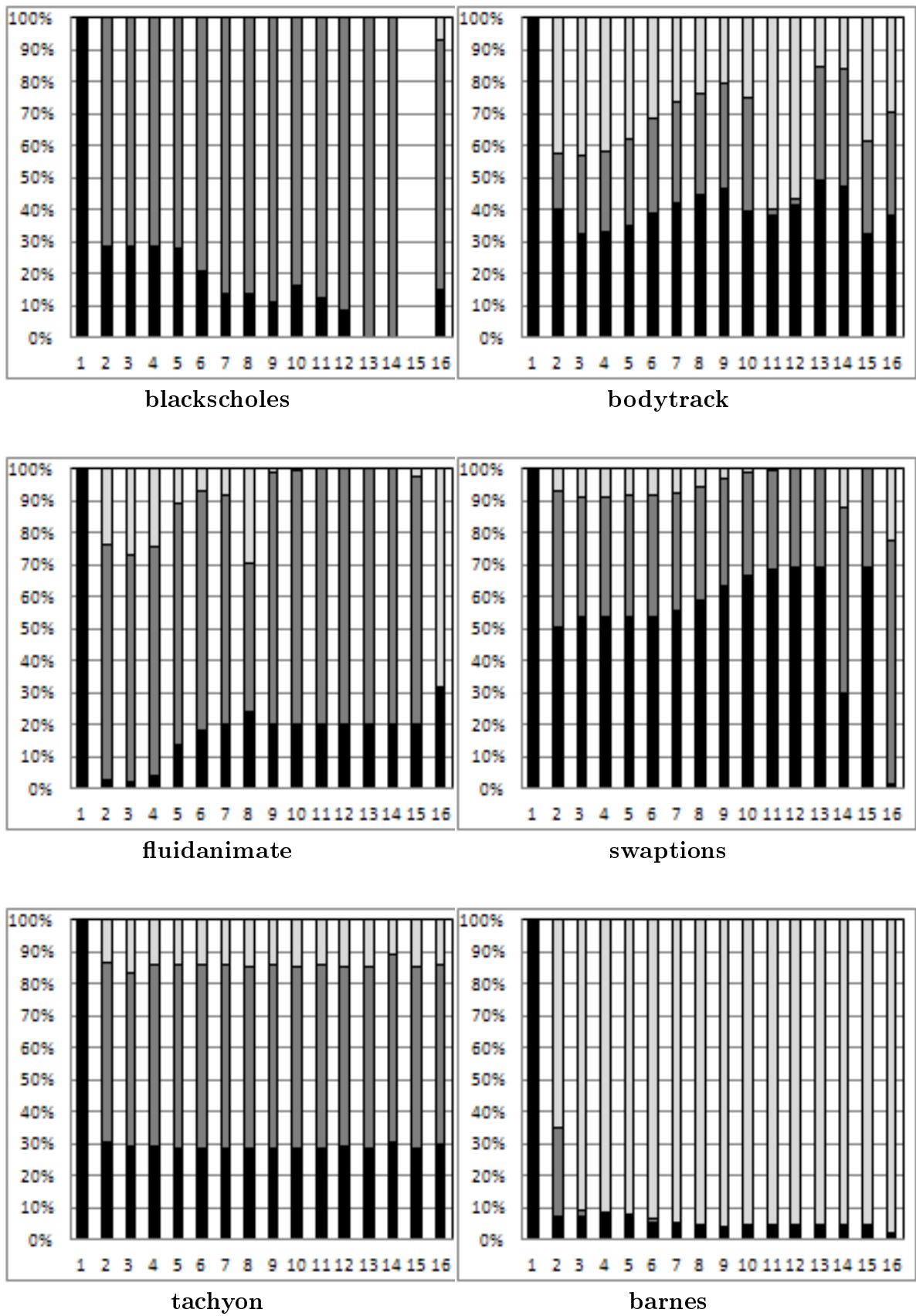


Figure 5.17. Memory access pattern (1/2)

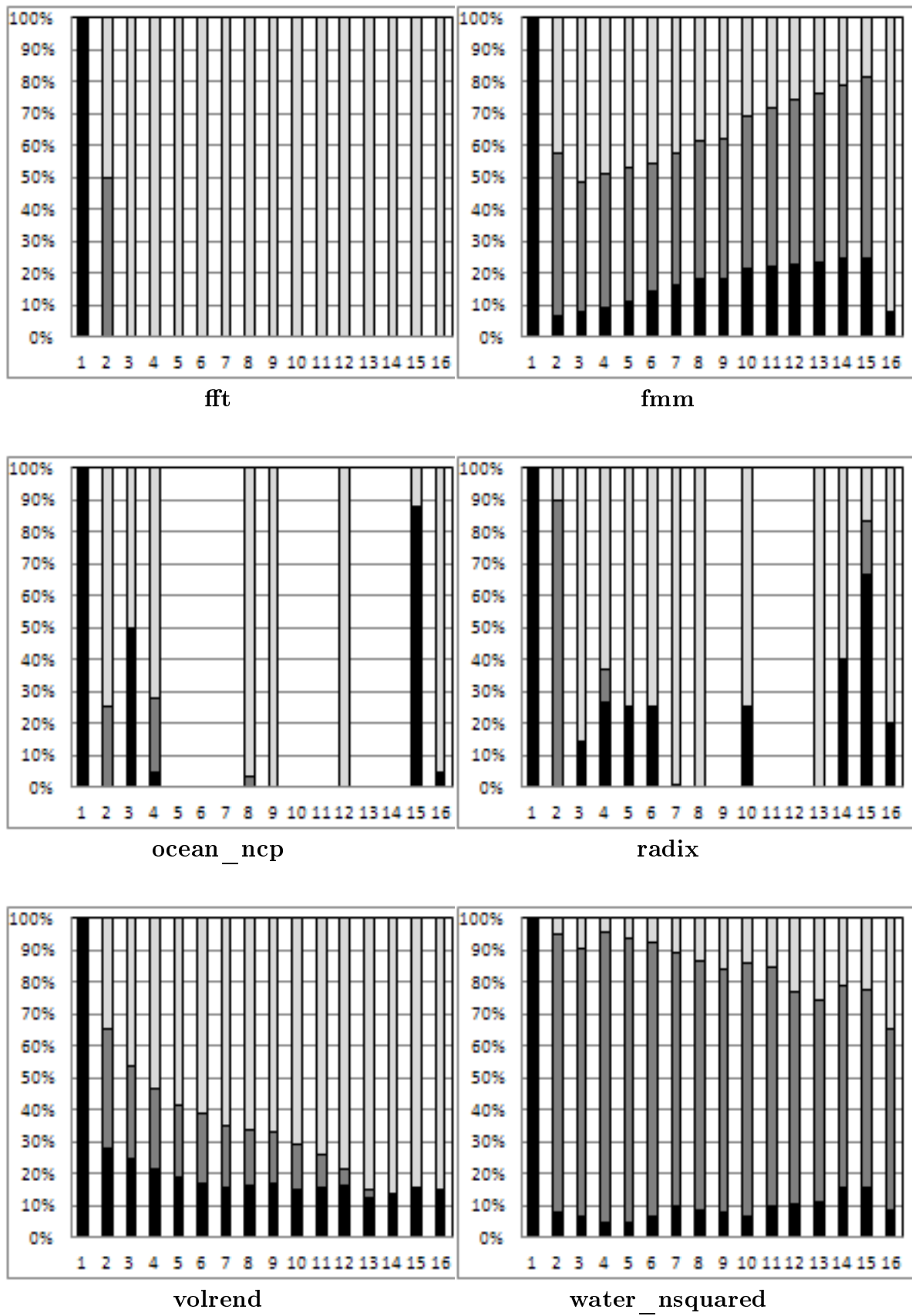


Figure 5.18. Memory access pattern (2/2)

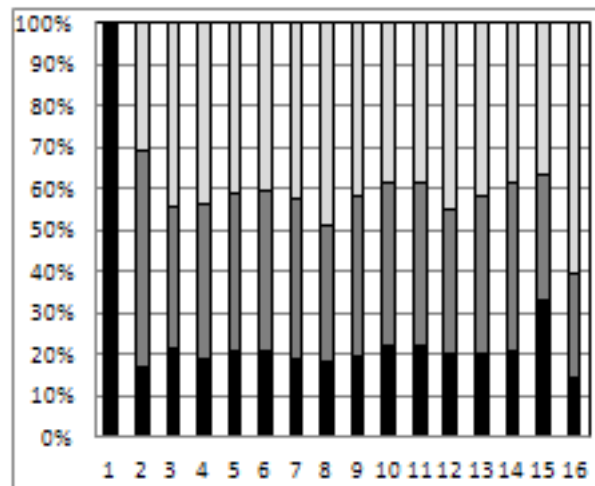


Figure 5.19. Average of memory access patterns

### 5.3.1 Discussion

We found a large amount of regularity in the memory access patterns. In the average with 16 active threads, 14.459% of the simultaneous memory accesses is uniform and 24.948% is affine. And in the total average, 25.446% of the accesses is uniform and 35.005% is affine. The amount of scattered accesses is high due to the allocation of data on the heap.

Data placed on the heap is not layer out in any organized pattern. The benchmarks and our experiments are not able to find patterns with heap data, because the benchmarks were written to run in machines like standard PC. Dynammmic memory allocation generally are not present in SIMD architecture, for instance, OpenCL, a language for GPGPU, does not support *malloc* and *free*.

As explained in 1.5.4, memory access regularity is good, because it allows many possible optimizations. Uniform address would cause only one shared access to the memory. Contiguous affine accesses can be optimized to be part of the same cache page and the accesses can be combined into a single memory transaction.

## 5.4 Distance between memory accesses

Here we show the distances between affine accesses and the distance that is the interval that includes all scattered accesses. Knowing the distances is important, because it gives the computer architect the subsidies to optimize memory access. Short distances allow the same memory access to retrieve data for different threads. GPUs have been using this technique extensively: threads that access nearby data have the opportunity to read or write these data through one single coalesced access.

In the charts, each number, e.g., 0 to 63, is the base-2 logarithm of the maximum distance between two addresses. We grouped the exponent values, from the darkest to the lightest, in 0 to 3, 4 to 7, 8 to 15, 16 to 31 and 32 to 64. For each benchmark, the left chart shows affine accesses and the right chart shows scattered accesses. The X axis is the number of active threads.

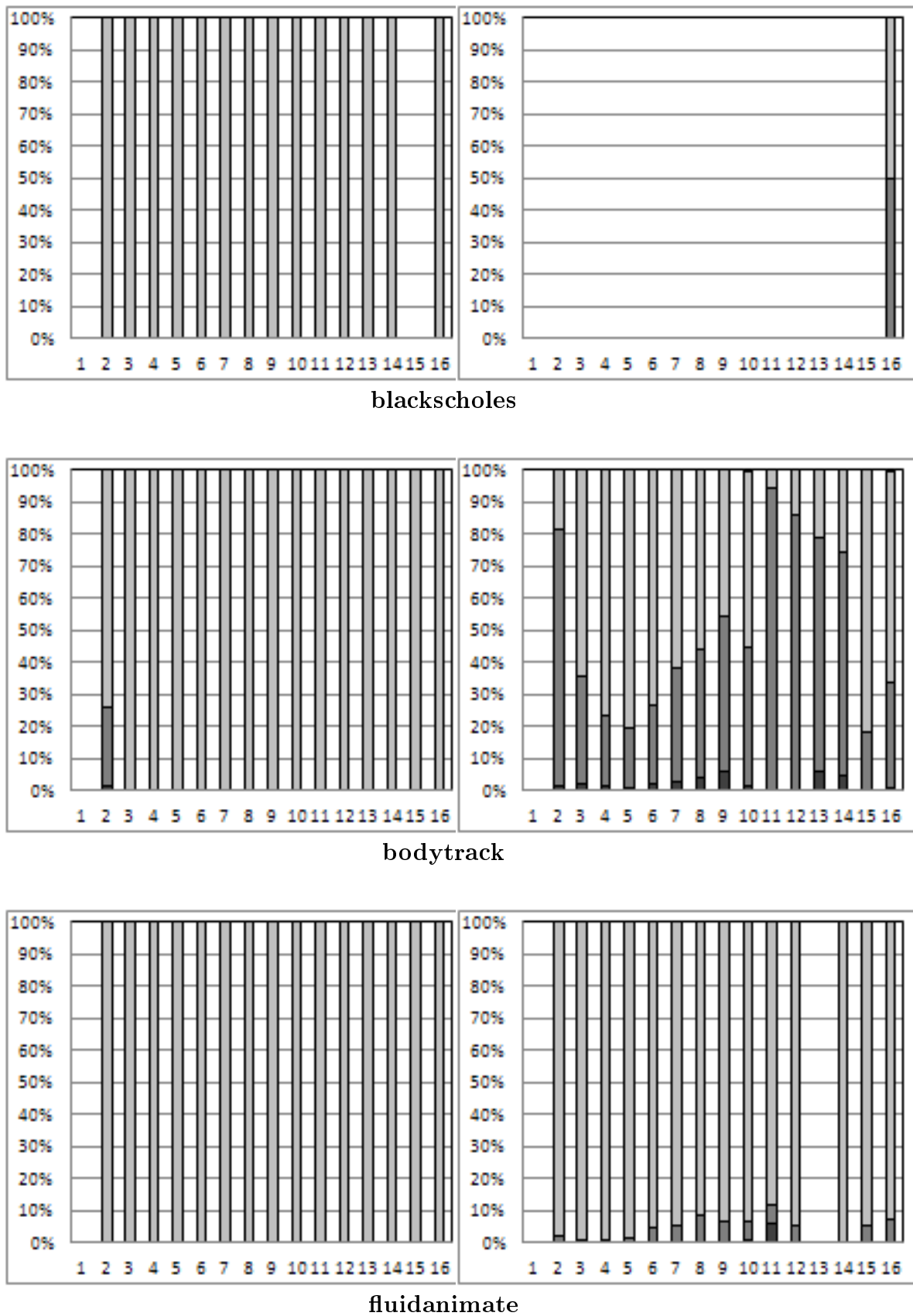


Figure 5.20. Memory access distances (1/4)

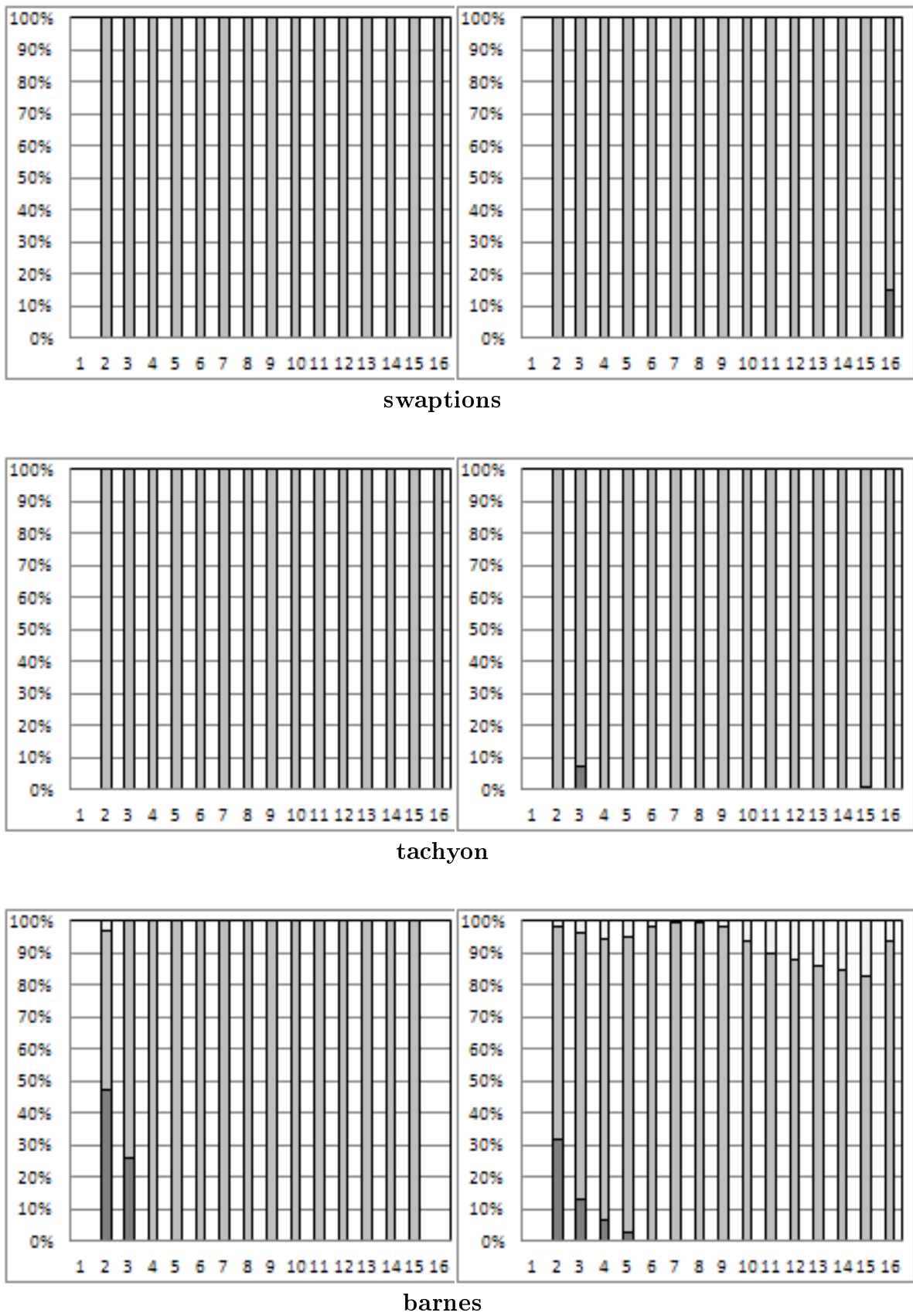


Figure 5.21. Memory access distances (2/4)

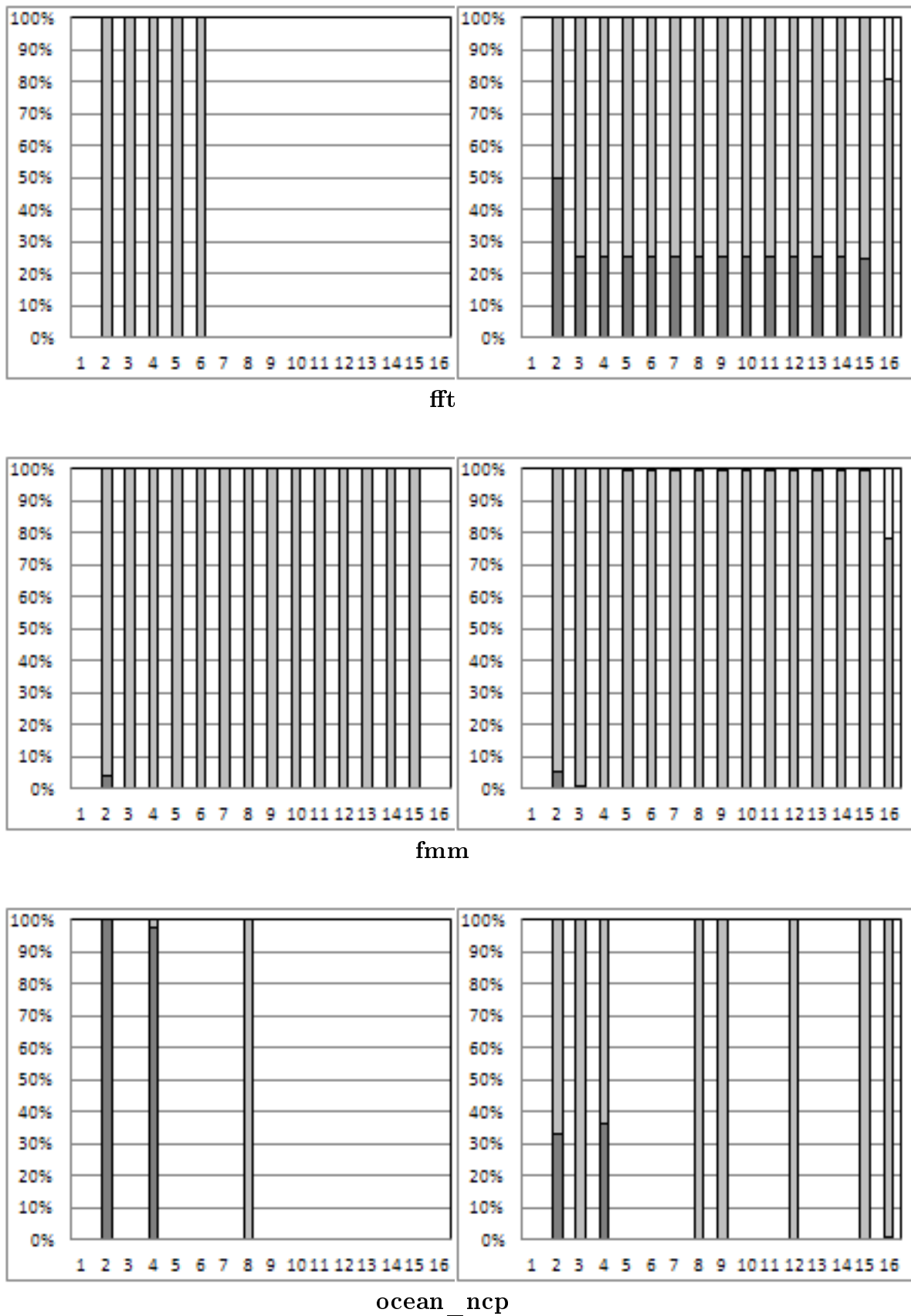


Figure 5.22. Memory access distances (3/4)



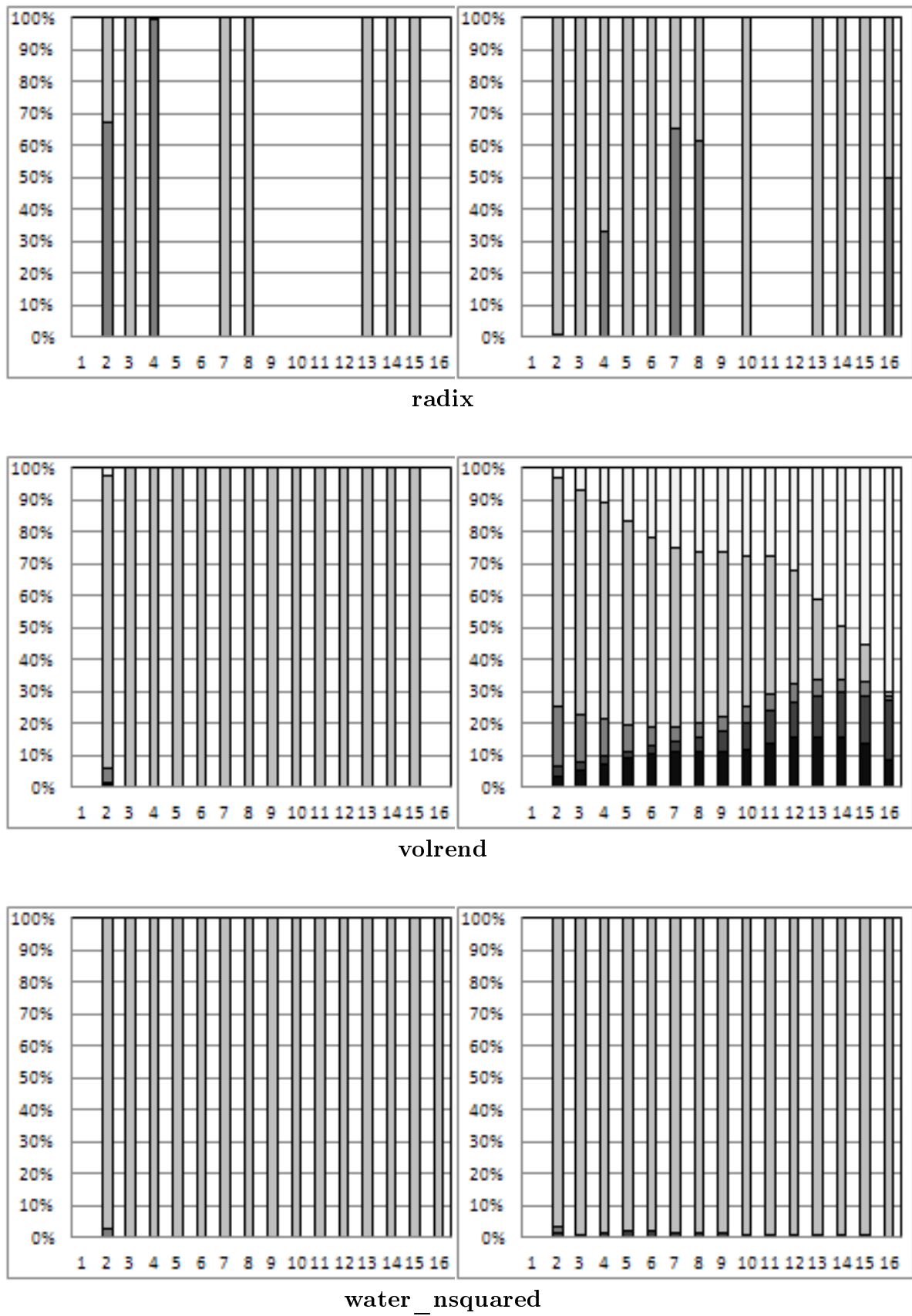


Figure 5.23. Memory access distances (4/4)

### 5.4.1 Discussion

The large quantity of affine accesses that we have observed is attributed to the stack frames. The most common distance between affine accesses is  $2^{23}$  bytes. Each thread receives  $2^{23}$  bytes of memory to allocate their stack frames. Thus, local variables, which are stored in the stack, are likely to be spaced  $2^{23}$  bytes.

We have not observed very close affine accesses, e.g., within blocks up to  $2^7$  bytes long. This behavior was expected because, in the current architecture, the blocks belongs to the same cache page. If multiple threads could write on it simultaneously, then we would experiment a performance penalty in PC architecture due to cache coherence updates.

Future compilers and architectures could merge the call stack of all threads into a single vectorial call stack. In the merged stack, each variable would be a vector with an instance per thread. Thus, the data that is stored in stack of different threads would be contiguous and it helps to optimize cache operations and memory access.

In general we have observed very long distances between the scattered addresses used by threads when simultaneously processing load and store instructions. Most distances that we have observed are between  $2^{16}$  and  $2^{31}$ . The data in this case is stored either in static memory, or, more usually, in the memory heap. The memory heap is often used to keep data structures of objects and containers.

We speculate that these large spaces between addresses are common because the benchmarks and compilers have not been done with memory coalescing in mind. They are meant to run in traditional CPUs, and in this case inter-thread locality is not at a premium. On the contrary, close inter-thread locality would be harmful in the context of multi-core platforms with coherent private caches, by causing false sharing of cache lines. Collange has observed a substantially different behavior in GPGPU applications [11]. In that case, inter-thread proximity is much more common, as this type of locality contributes notoriously to performance improvements.

# Chapter 6

## Final Remarks

In this dissertation we have advanced the research on minimal multi-threaded hardware. We believe that this is an important contribution, because parallelism is, currently, the main avenue towards high-performant systems. Our work paves the way to multi-threaded processors to run data-parallel applications as efficiently as possible.

We have compared different thread-synchronization heuristics. The MinSP-MinPC and MaxFun-MinPC heuristics were proposed during this study and our empirical evaluation has demonstrated that these techniques are very effective in keeping threads synchronized. In addition to being effective, we claim that MinSP-MinPC and MaxFun-MinPC are one of the simplest heuristics available in the literature.

We have also studied the memory access patterns typical of data-parallel multi-threaded applications. This study let us find a substantial amount of regularity between concurrent threads. This regularity is a further motivation for new hardware designs that have been proposed in the literature, but are yet to be manufactured. New hardware designs might include better memory coalescing, uniform memory access sharing and merged call stack.

Finally, we have analysed the distance between the addresses of data accessed simultaneously by different threads. Data accessed by different threads tend to be distant in memory. This distance makes it difficult to take benefit from spatial locality in inter-thread memory accesses. This somehow negative result suggests that the data layout of the call stack should be reconsidered in the context of inter-thread locality. We leave this new study as future work.

## 6.1 Limitations of this work

We have not succeed to find a heuristic that is better than all the others in every scenario. Although we identified some specific cases to be optimized, no heuristic that is good in all cases was found. We also show that the current state-of-art priority policies can be improved.

Our experiments were executed with full benchmarks. We should have divided the study to find good heuristics in two steps. The first part would use programs with single function with any kind of realistic and complex CFG. It would help to find good heuristics for intra-function synchronization. The second step would include programs with complex graph of function calls.

Another limitation of this work is that we have tested all our techniques in software only. We have not simulated them in hardware. A deeper study, involving possibly a cycle accurate hardware simulator or FPGA implementation, would let us probe the real cost of each heuristic in terms of wires, gates and delays. Although we have not studied the cost of the real hardware implementation, our work has allowed us to verify that very good synchronization policies exist. This positive result leads us to think that better heuristics with cheap hardware implementation may come in the future.

From this last observation, we believe that our post-dominator heuristics can be improved in a real hardware implementation. We have implemented these heuristics in software, using sequential algorithms to compare PCs. Our simulation in software relies on many compiler hints that were inserted in the code increasing the size of the code. We believe that this heuristics has great potential to benefit from new helper instructions and specific compiler optimizations.

## 6.2 Future works

Throughout this work, we had many ideas that could be used to create heuristics, but we did not implement them for the lack of time. One of these ideas is to use the branch predictor, which may be already available in hardware, to help in the synchronization effort. The branch predictor may help the synchronizer to guess where a *break* or *continue* statements of a loop is. In other words, it may detect when a thread finishes a loop, or an iteration within this loop. Thus, we may give priority to threads that have not finished the current iteration or have not finished the loop yet, forcing threads that are ahead in the loop to wait.

Since we do not have a heuristic that is good for all cases, as a future work, an

adaptive heuristic could be created. It would be able to detect the behavior of the program and choose a known heuristic to handle it. It may also analyse characteristics of specific parts the program, to decide the best heuristic to synchronize it. Its hardware implementation is a challenge, because it is very expensive.

A natural continuation of our work is to investigate heuristics that are good at synchronizing threads at the beginning of called functions. Some heuristics, such as MaxFun-MinPC, manage to synchronize the threads when the functions return. However, it does not do anything when the same function is reached through different intermediate function calls. We believe that handling this case is an important contribution to this work, because it tends to improve the performance of object-oriented programs that run in parallel. In this world, we have many virtual function calls, which may even be invoked through different names and small overridden functions may call the corresponding function of the parent class. Ideally, a good heuristic should be able to synchronize threads with many levels of function calls.

For instance, in figure 3.5, we have the class *Animal*, which is inherited by *Wolf* and *Tiger*. Both classes override the function *Animal::eat* and both *Wolf::eat* and *Tiger::eat* call *Animal::eat*. In this example, we have three threads that called the virtual function *instance->eat()* and the first thread called *Wolf::eat*, the second one called *Tiger::eat* and the third one called *Animal::eat* directly. We would like to make all threads execute *Animal::eat* simultaneously.

Another case is present in figure 3.1. In this case, we would like to make all threads execute *big\_function* simultaneously without worsening the heuristics. This specific problem is solved by branch fusion [16], which is a technique that divides "if-then-else" in parts and redundant code is not repeated. However, we can also have a case of "if-then-else" in which *then* calls function *foo*, *else* calls function *bar*, and both functions are small and call function *fubaz*, which we want to synchronize.

We have not got good results with the policy of the longest path. New algorithms to solve with better accuracy the longest path distance in cyclic graphs could be invented in the future. The algorithm would be optimized for CFG, which has one or two edges per node. The exit node has no out edge and is the only destination, whose longest distance from any node is desired. Sometimes, a large CFG may have small isolated subgraphs that may be solved locally with non-polynomial-time algorithm. For instance, we have a subgraph between the entry node, which must have no predecessor, and the nearest post-dominator of the entry node; and no node can have a predecessor that is not part of the subgraph. The solution of the subgraph helps to decrease the size of the full CFG.

We believe that the linker can be used to help existing heuristics and to design

new heuristics. We envision an algorithm in which the linker creates a graph of the entire program where each function is a node and edges link caller to callee. This graph starts with the first function that the threads execute. The linker can sort topologically the graph. The linker could add priority hints or synchronization hints in the beginning of each function to help when the same functions is reached through different paths. The linker may also sort all the functions of the program favorably to help the Min-PC heuristic. Notice that implementing such an algorithm is challenging. The optimization of programs with indirect calls, including virtual functions, would be difficult because they cannot be easily included in the graph, but the linker can see the entire program. Functions of shared libraries would be a further problem.

As a future work, we would also like to implement the heuristic that uses structured control flow graph with *break* and *continue* statements, which is explained in 3.2.2. It requires a structured program, but it can be compiled using an algorithm to transform the input CFG into a structured CFG. This transformation leads to code expansion. However, we speculate that this kind of heuristic may be very effective at synchronizing threads; thus, leading to shorter execution traces.

Another work that we leave for the future involves the study of architectures between SIMD and MIMD, such as 2IMD and 4IMD. The study of these architectures has been done previously; however, the literature has yet to compare and find good heuristics for them. TLP is always 1 in SIMD, but it varies from 1 to 2 in 2IMD and from 1 to 4 in 4IMD. When the DLP of some benchmarks cannot be improved, the TLP should be higher to improve performance. Implementing these heuristics is also a difficult endeavor. A good implementation may require the use of compiler support and changes in ISA. These changes should give us the opportunity to decide if a thread should execute the second path of a divergent block, or if it should wait for synchronization. This decision depends on the code structure, e.g., are we in an "if-then-else block" or are we in an "if-then block"?

Finally, an architecture that is intermediate between SIMD and MIMD could emerge. A future study should find a good number of fetch units in relation to the number of execute units that a parallel machine should have to run data-parallel programs. This number may equal the largest number of divergent paths in the program. In programs whose DLP is 2 and TLP is high, an hIMD architecture would have good performance. The hIMD would be an architecture in which the number of fetch units is half of the number of execute units,

## 6.3 Software

We expect that our contributions may help computer engineers to create new concrete hardware that is faster, cheaper and more power-efficient. Towards this end, we leave more than empirical results in this work. We have developed a framework to create and execute new heuristics. We believe that it may be useful to other researchers in the future. The source code of this framework, of the heuristics and of the many benchmarks that we have developed during our toils can be found in: <http://code.google.com/p/mmt-sync/>





# Bibliography

- [1] Acar, U., Bletloch, G., and Harper, R. (2002). Adaptive functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*.
- [2] Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5:1–19.
- [3] Azam, M., Franzon, P., and Liu, W. T. (1997). Low power data processing by elimination of redundant computations. In *International Symposium on Lower Power Electronics and Design*.
- [4] Barone, P., Bonizzoni, P., Vedova, G. D., and Mauri, G. (2001). An approximation algorithm for the shortest common supersequence problem: an experimental analysis. In *SAC*, pages 56–60. ACM.
- [5] Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.
- [6] Biswas, S., Franklin, D., Savage, A., Dixon, R., Sherwood, T., and Chong, F. T. (2009). Multi-execution: Multi-core caching for data similar executions. In *International Symposium on Computer Architecture*.
- [7] Brunie, N., Collange, S., and Damos, G. (2012). Simultaneous branch and warp interweaving for sustained GPU performance. In *ISCA*, pages 49–60. ACM.
- [8] Chakraborty, K., Wells, P. M., and Sohi, G. S. (2006). Computation spreading: Employing hardware migration to specialize cmp cores on-the-fly. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [9] Citron, D., Feitelson, D., and Rudolph, L. (1998). Accelerating multi-media processing by implementing memoing in multiplication and division units. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

- [10] Citron, D. and Feitelson, D. G. (2000). Hardware memorization of mathematical and trigonometric functions. Technical report 2000-5, Hebrew University of Jerusalem.
- [11] Collange, S. (2011). Stack-less SIMT reconvergence at low cost. Technical report, ENS Lyon.
- [12] Collange, S., Defour, D., and Zhang, Y. (2009). Dynamic detection of uniform and affine vectors in GPGPU computations. In *HPPC*, pages 46–55. Springer.
- [13] Collange, S. and Kouyoumdjian, A. (2011). Affine vector cache for memory bandwidth savings. Technical report ensl-00649200, ENS Lyon.
- [14] Coon, B. W. and Lindholm, J. E. (2008). System and method for managing divergent threads in SIMD architecture. US Patent 7353369.
- [15] Costa, A. T., Franca, F. M. G., and Filho, E. M. C. (2000). The dynamic trace memorization reuse technique. In *International Conference on Parallel Architectures and Compilation Techniques*.
- [16] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2011). Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE.
- [17] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2012). Profiling divergences in GPU applications. *Concurrency and Computation: Practice and Experience*, 1(10.1002/cpe.285-15):1–15.
- [18] Dechene, M., Forbes, E., and Rotenberg, E. (2010). Multithreaded instruction sharing. Technical report, Department of Electrical and Computer Engineering, North Carolina State University.
- [19] Damos, G., Kerr, A., Wu, H., Yalamanchili, S., Ashbaugh, B., and Maiyuran, S. (2011). SIMD re-convergence at thread frontiers. In *MICRO*.
- [20] Damos, G., Kerr, A., Yalamanchili, S., and Clark, N. (2010). Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, pages 354–364. IEEE.
- [21] Fung, W. and Aamodt, T. (2011). Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36.

- [22] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. (2007). Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, Washington, DC, USA. IEEE Computer Society.
- [23] Golander, A. and Weiss, S. (2009). Reexecution and selective reuse in checkpoint processors. In *Transactions on High-Performance Embedded Architectures and Compilers*, pages 242–268.
- [24] Gonzalez, A., Tubella, J., and Molina, C. (1998). The performance potential of data value reuse. Technical report UPC-DAC-1998-23, UPC.
- [25] González, J., Cai, Q., Chaparro, P., Magklis, G., Rakvic, R., and González, A. (2008). Thread fusion. In *ISLPED*, pages 363–368. ACM.
- [26] Han, T. D. and Abdelrahman, T. S. (2011). Reducing branch divergence in GPU programs. In *GPGPU-4*, pages 3:1--3:8. ACM.
- [27] Harizopoulos, S. and Ailamaki, A. (2004). Steps towards cache resident transaction processing. In *30th Very Large Database (VLDB) conference*.
- [28] Hennessy, J. L., Patterson, D. A., and Arpaci-Dusseau, A. C. (2007). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- [29] Intel (2009). *Intel G45 Express Chipset Graphics Controller PRM, Volume Four: Subsystem and Cores*. Intel.
- [30] Intel (2010). *Intel HD Graphics OpenSource PRM Volume 4 Part 2: Subsystem and Cores - Message Gateway, URB, Video Motion, and ISA*. Intel.
- [31] Kerr, A., Diamos, G. F., and Yalamanchili, S. (2009). A characterization and analysis of GPGPU kernels. In *IISWC*, pages 3–12. IEEE.
- [32] Keryell, R. and Paris, N. (1993). Activity counter: New optimization for the dynamic scheduling of SIMD control flow. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 184–187.
- [33] Kumar, R., Jouppi, N. P., and Tullsen, D. M. (2004). Conjoined-core chip multiprocessing. In *IEEE/ACM International Symposium on Microarchitecture*, pages 195–206.
- [34] Lashgar, A. and Baniasadi, A. (2011). Performance in GPU architectures: Potentials and distances. In *WDDD*, pages 75–81. IEEE.

- [35] Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.
- [36] Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, C., and Asanovic, K. (2011). Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ISCA*, pages 129–140. ACM.
- [37] Levinthal, A. and Porter, T. (1984). Chap - a SIMD graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82.
- [38] Lindholm, J. E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55. ISSN 0272-1732.
- [39] Long, G., Franklin, D., Biswas, S., Ortiz, P., Oberg, J., Fan, D., and Chong, F. T. (2010). Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors. In *MICRO*, pages 337–348. IEEE.
- [40] Lorie, R. A. and Strong, H. R. (1984). Method for conditional branch execution in SIMD vector processors. US Patent 4435758.
- [41] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). PIN: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, Chicago, IL.
- [42] Malits, R., Mendelson, A., Kolodny, A., and Bolotin, E. (2012). Exploring the limits of GPGPU scheduling in control flow bound applications. *HIPEAC*, 8(29).
- [43] Meng, J., Tarjan, D., and Skadron, K. (2010). Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246. ISSN 0163-5964.
- [44] Milanez, T., Collange, S., Pereira, F. M. Q., Jr., W. M., and Ferreira, R. (2012). Data and instruction uniformity in minimal multi-threading. In *SBAC-PAD*, pages 270–277.
- [45] Molina, C., Gonzalez, A., and Tubella, J. (1999). Dynamic removal of redundant computations. In *International Conference of Supercomputing*.
- [46] Mutlu, O., Kim, H., Stark, J., and Patt, Y. N. (2005). On reusing the results of pre-executed instructions in a runahead execution processor. In *IEEE Computer Architecture Letters*.

- [47] Narasiman, V., Lee, C. J., Shebanow, M., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N. (2011). Improving GPU performance via large warps and two-level warp scheduling. In *MICRO 44: Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*.
- [48] Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30:56–69. ISSN 0272-1732.
- [49] Oh, J., Hwang, S. J., Nguyen, H. G., Kim, A., Kim, S. W., Kim, C., and Kim, J.-K. (2008). Exploiting thread-level parallelism in lockstep execution by partially duplicating a single pipeline. *ETRI Journal*, 30(4).
- [50] Parashar, A., Gurusurthi, S., and Sivasubramaniam, A. (2004). A complexity effective approach to alu bandwidth enhancement for instruction level temporal redundancy. In *International Symposium on Computer Architecture*.
- [51] Patterson, D. A. and Hennessy, J. L. (2009). *Computer organization and design: the hardware/software interface*. Morgan Kaufmann.
- [52] Pilla, M. L., Navaux, P. O. A., and Childers, B. R. (2004). Value predictors for reuse through speculation on traces. In *Symposium on Computer Architecture and High Performance Computing*.
- [53] Quinn, M. J., Hatcher, P. J., and Jourdenais, K. C. (1988). Compiling C\* programs for a hypercube multicomputer. *SIGPLAN Not.*, 23:57–65.
- [54] Richardson, S. E. (1992). Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report SMLI TR-92-1, Sun Microsystems Laboratories.
- [55] Richardson, S. E. (1993). Exploiting trivial and redundant computation. In Swartzlander, E. E., Irwin, M. J., and Jullien, J., editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, Windsor, Canada. IEEE Computer Society Press, Los Alamitos, CA.
- [56] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and mei W. Hwu, W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using cuda. In *PPoPP*, pages 73–82. ACM.
- [57] Sampaio, D., Martins, R., Collange, S., and Pereira, F. M. Q. (2012). Divergence analysis with affine constraints. In *SBAC-PAD*, pages 137–146. IEEE.

- [58] Schrijver, A. (2003). *Combinatorial Optimization: Polyhedra and Efficiency, Volume 1, Algorithms and Combinatorics*. Springer. ISBN 9783540443896.
- [59] Seznec, A. and Espasa, R. (2005). Conflict-free accesses to strided vectors on a banked cache. *IEEE Transactions on Computers*, 54:913–916.
- [60] Sodani, A. and Sohi, G. S. (1997). Dynamic instruction reuse. In *International Symposium on Computer Architecture*.
- [61] Sodani, A. and Sohi, G. S. (1998). An empirical analysis of instruction repetition. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [62] Stone, J. E. (2007). Tachyon raytracer. Raytracer and benchmark.
- [63] Surendra, G., Banerjee, S., and Nandy, S. K. (2003). Enhancing speedup in network processing applications by exploiting instruction reuse with flow aggregation. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*.
- [64] Takahashi, Y. (1997). A mechanism for SIMD execution of SPMD programs. In *Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97, HPC-ASIA '97*, pages 529–534.
- [65] Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23:392–403.
- [66] University of Washington (2006). Splash-2. Benchmark package.
- [67] Wang, W. D. and Raghunathan, A. (2004). Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization. In *International Conference on VLSI Design*.
- [68] Weinberg, N. and Nagle, D. (1998). Dynamic elimination of pointer-expressions. In *International Symposium on Parallel Architectures and Compilation Techniques*.
- [69] Wu, H., Diamos, G., Li, S., and Yalamanchili, S. (2011). Characterization and transformation of unstructured control flow in GPU applications. *The First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*.
- [70] Yang, Y., Xiang, P., Kong, J., and Zhou, H. (2010). A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97. ACM.

- [71] Zhang, F. and H. D'Hollander, E. (2004). Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, 30(4):231–245. ISSN 0098-5589.