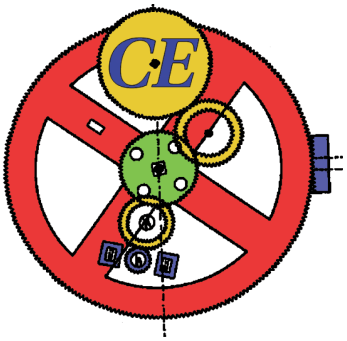


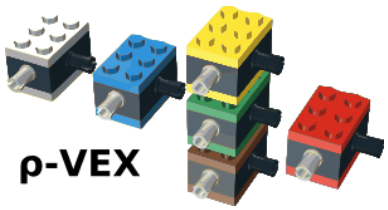
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

Task Scheduling for Adaptive Reconfigurable VLIW Multicore Processors

Georgios Andronikidis



CE-MS-2014-16

Abstract

Embedded Reconfigurable Architectures (ERA) is a project with the objective to design a platform that combines reconfigurable computing and network elements which can adapt on-the-fly their composition, organization and even instruction-set architectures in an effort to provide the best possible trade-offs in performance and power for the given application(s). Although some of this adaptiveness is controlled by software (mainly the operating system), the goal of the ERA project is that great deal of this control actually takes place automatically at hardware level, by the Hardware scheduler. This thesis deals specifically with the problem of hardware task scheduling. We studied a variety of possible implementations for the task scheduling in the ERA platform. After getting an inside look of the Processing component of the ERA platform and understanding the particularities of it, as well as of its main building block, the ρ-VEX core, we tried to find scheduling algorithms available in the bibliography to implement as the Hardware scheduler of ERA. This literature research did not yield any results, both because of the complexity of the problem, as well as the pioneer characteristics of ERA that we would like to take advantage of. We designed some simple scheduling algorithms, especially tailored for the ERA platform and tested them. The most important of them were Basic, which simply stalls the tasks until there are enough resources for them, Versioning, which brings a different binary from the memory which is compiled to run on a smaller core and Generic Binary which uses a binary that is especially compiled to run on any core, so downgrading a task does not lead to the communication cost that Versioning suffers and upgrading a task becomes possible. We present the most important of the experiments that took place within this thesis and show that GB++ (a version of GB that supports forced priorities, interrupts and upgrading by default) is the most promising algorithm that can take advantage of all the characteristics and the abilities of ERA, without being the fastest one, which is Versioning. Finally, we researched and defined the minimum requirements of GB++ in order to become apart from the rest also the fastest algorithm for ERA.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Task Scheduling for Adaptive Reconfigurable VLIW Multicore Processors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Georgios Andronikidis
born in Ptolemais, West Macedonia, Greece

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

Task Scheduling for Adaptive Reconfigurable VLIW Multicore Processors

by Georgios Andronikidis

Abstract

Embedded Reconfigurable Architectures (ERA) is a project with the objective to design a platform that combines reconfigurable computing and network elements which can adapt on-the-fly their composition, organization and even instruction-set architectures in an effort to provide the best possible trade-offs in performance and power for the given application(s). Although some of this adaptiveness is controlled by software (mainly the operating system), the goal of the ERA project is that great deal of this control actually takes place automatically at hardware level, by the Hardware scheduler. This thesis deals specifically with the problem of hardware task scheduling. We studied a variety of possible implementations for the task scheduling in the ERA platform. After getting an inside look of the Processing component of the ERA platform and understanding the particularities of it, as well as of its main building block, the ρ-VEX core, we tried to find scheduling algorithms available in the bibliography to implement as the Hardware scheduler of ERA. This literature research did not yield any results, both because of the complexity of the problem, as well as the pioneer characteristics of ERA that we would like to take advantage of. We designed some simple scheduling algorithms, especially tailored for the ERA platform and tested them. The most important of them were Basic, which simply stalls the tasks until there are enough resources for them, Versioning, which brings a different binary from the memory which is compiled to run on a smaller core and Generic Binary which uses a binary that is especially compiled to run on any core, so downgrading a task does not lead to the communication cost that Versioning suffers and upgrading a task becomes possible. We present the most important of the experiments that took place within this thesis and show that GB++ (a version of GB that supports forced priorities, interrupts and upgrading by default) is the most promising algorithm that can take advantage of all the characteristics and the abilities of ERA, without being the fastest one, which is Versioning. Finally, we researched and defined the minimum requirements of GB++ in order to become apart from the rest also the fastest algorithm for ERA.

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-16

Committee Members :

Advisor:	Dr. ir. Stephan S. Wong, CE, TU Delft
Chairperson:	Dr. ir. Stephan S. Wong, CE, TU Delft
Member:	Dr. Zaid Al-Ars, CE, TU Delft
Member:	Dr. ir. T.G.R.M van Leuken, CE, TU Delft

To my beloved parents and in memory of Vasiliki Theodoridou:

«Της Βάσως τα εγγόνια να λένε...»
("So that people talk about Vaso's grandchildren...")

Contents

List of Figures

List of Tables

List of Acronyms

FPGA Field-Programmable Gate Array

ERA Embedded Reconfigurable Architectures

VLIW Very Long Instruction Word

MCP Multicore Processor

VEX VLIW Example

ISA Instruction Set Architecture

ILP Instruction Level Parallelism

DLP Data Level Parallelism

NOP No Operation

NOC Network on Chip

FIFO First In First Out

OS Operating System

AlgA Algorithm A

RAlgA Realistic Algorithm A

RAlgAW8 Realist Algorithm A, Window=8

GB Generic Binary

AlgB Algorithm B (Generic Binary)

AlgBall8 Algorithm B, all tasks are 8-issue

AlgD Alogirthm D

AlgDall8 Alogirthm D, all tasks are 8-issue

List of Algorithms

Acknowledgements

First of all, I would like to thank my supervisor, Stephan Wong, for the opportunity he gave me to be a small part of the ERA project and his guidance throughout the whole thesis. For his patience and his understanding as well as the way he handled my particular character. The only thing I will never forgive him for, is making me learn and use LaTeX.

Of course Fakhar Anjam and Anthony Brandon for their unconditional help and support. TU Delft is only lucky having such helpful and exceptional researchers.

Especially I want to thank my family, who stood by my side during the whole period of my studies, in its goods and its bads, as well as Bram Masseur and Massimiliano Marass, who did their best to keep all my distractions away during the period of this thesis and generally all my friends who tried to give me courage and energy.

I am also very grateful to Inge Verhoeven and John Stals, without the support of whom I might have never undertaken this project.

Finally, although he can't read these lines any more, I want to thank Stamatis Vassiliadis, for making everything he could for me to come to TU Delft.

A very special thanks to all those small and big scientists that doubt, question and challenge "reality" and "truth" constantly; for they are the true pioneers of science and my inspiration.

Georgios Andronikidis
Delft, The Netherlands
November 5, 2014

1 Introduction

When reconfigurable hardware was invented (especially Field-Programmable Gate Arrays - FPGAs) a new era began for Computer Engineering. Custom solutions could be designed for specific customers without the need of ordering millions of copies to make each solution affordable, as is the case with wafer solutions. Research in small scale and budget also became possible, spin-off companies' pop-up boosted. What made reconfigurable hardware even more attractive was the possibility of an upgrade. A better version of the hardware without making the old purchase useless, incredibly handy both for research and industry.

Adaptive hardware changed the rules of hardware design for good; it was a great innovation. But even more amazing opportunities arise the last years that technologies are invented which allow more and more the hardware to adapt on-the-fly. No need to switch off the system, re-arrange the hardware and switch it back on. Designers naturally start thinking of the possibilities that arise when you can design a platform the hardware of which can change any moment one wants, without putting the system down. This is how the ERA project was born.

1.1 The ERA project

The idea behind ERA (Embedded Reconfigurable Architectures) is to design a platform that combines reconfigurable computing and network elements which can adapt on-the-fly their composition, organization and even instruction-set architectures in an effort to provide the best possible trade-offs in performance and power for the given application(s). On ERA, network elements and topologies as well as memory hierarchy organization can be selected both statically at design time and dynamically at run-time [1]. That last one is what makes ERA special, because the hardware adapts to the software that runs on it on every cycle. All three basic hardware components of the ERA platform (Processing component, Network component and Memory component) shown in Figure 1.1 are actually dynamically adaptable. Although some of this adaptiveness is controlled by the software (mainly the operating system), the goal of the ERA project is that great deal of this control actually takes place automatically on hardware level, by the Hardware scheduler.

1.2 Motivation

This thesis deals with the problem of task scheduling. The Task scheduler is the part of the Hardware scheduler seen on Figure 1.1 which with help by the Monitoring hardware is responsible for the reconfiguration of the Processing components. We will try to explore

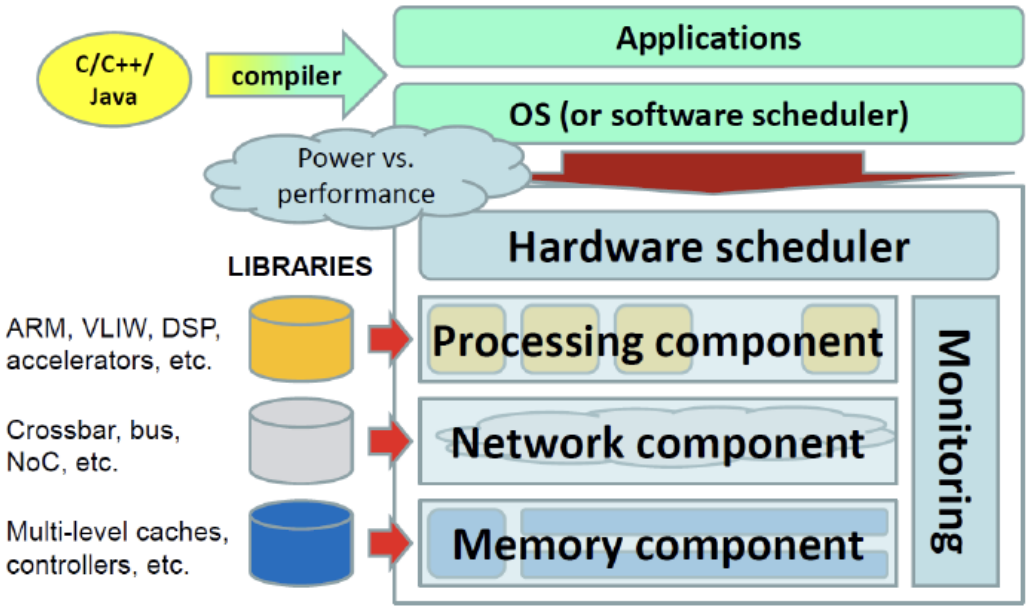


Figure 1.1: The ERA platform.

possible implementations/algorithms for the task scheduling in the ERA platform. That is, we will try to find or invent algorithms that can look at the several tasks which are being executed every moment and try to map them on hardware within one or in any case finite number of cycles. This is the core idea of the ERA project. If this is not feasible, then the ERA platform would just be yet another VLIW system.

1.3 Goals

The main goal of this thesis is to explore and compare the several possibilities/algorithms for task scheduling on the ERA platform. If no appropriate algorithms are available in the literature, we will have to invent some. Those algorithms, wherever they come from eventually, will have to be tested and compete with each other. The results should be analysed and if there is time, the chosen algorithm could be implemented inside ERA to see if it confirms the results of the research.

1.4 Methodology

In the most abstract level the methodology that will be followed is described by the following steps:

- Define the problem.
- Set the scope of the research.

- Look in the bibliography for solutions.
- Invent own/custom solutions if necessary.
- Test solutions.
- Analyse the results and draw conclusions.

More specifically the methodology in every step is described by the Figure 1.2 below.

1.5 Overview

This thesis will be structured as follows. First, in Chapter 2 we are going to give some background information necessary for the understanding of the problem and the way we tried to tackle it, including the preliminary literature research performed. Then, in Chapter 3, the several algorithms will be presented and in Chapter 4 the simulator that we had to build to perform the research. In Chapter 5 the results of the research will follow and Chapter 6 concludes the thesis with a summary of all the conclusions drawn in the previous chapters, as well as some proposals for future work.

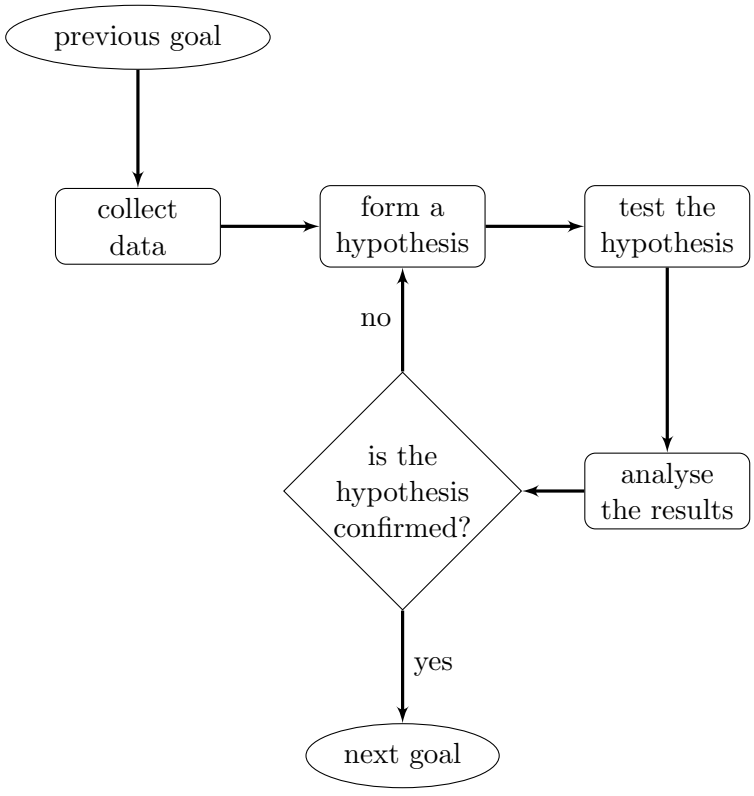


Figure 1.2: Flow chart showing of methodology for every goal.

2 Background

In this chapter we provide some background information about the ERA project as well as the literature research that took place in order to find out whether a solution that can be implemented in ERA and makes use of its special characteristics already exists.

2.1 The ρ-VEX VLIW processor

In the previous chapter, specifically on Figure 1.1, we saw an abstract block diagramme of the ERA platform. Since the focus of this thesis is on the Processing component seen in the figure, it is important to look inside that box more carefully. The goal is that the Processing component will be adapted constantly on the software running on the platform on any moment. That is why a Multicore Processor (MCP) described in [2] is implementing it, an abstract block diagramme of which is depicted in Figure 2.1 below. The originality lays on the fact that the core/cluster size and the number of cores/clusters can be adapted on-the-run.

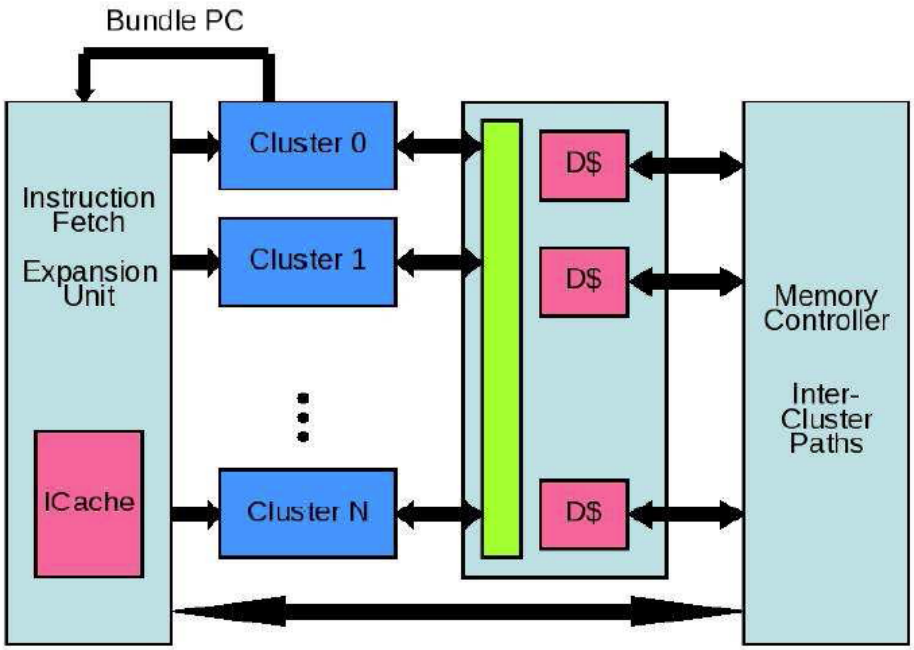


Figure 2.1: The block diagramme of a VEX Multicore Processor.

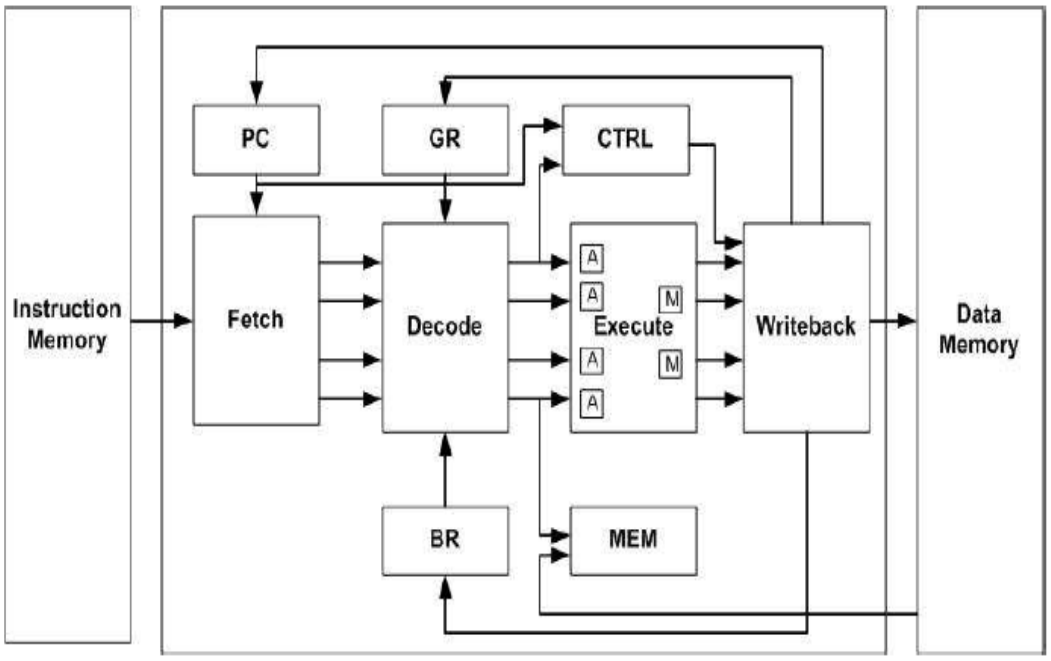


Figure 2.2: The block diagramme of a 4-issue ρ-VEX.

The smallest core of this Multicore Processor is a 2-issue ρ-VEX VLIW processor[3]. ρ-VEX is a VLIW processor that implements the VEX ISA[4]. It is a reconfigurable and parameterized processor[5], the most relevant parametre of which is the issue width. A 4-issue ρ-VEX for example is depicted in Figure 2.2 above.

Two clusters can be combined to form a bigger one. For example two 2-issue cores can be combined to form one 4-issue core/cluster or two 4-issue cores can be combined to form one 8-issue core or two 8-issue cores can be combined to form one 16-issue core and so on. Theoretically there is no limit, but practically (and especially on FPGAs) the resources are limited and so is the parallelism VLIW processors can exploit. There is no meaning in using a 16-issue core when we hardly ever can find 16 operations that can be executed simultaneously. In that case it would be wiser to use two 8-issue cores and let 2 tasks run simultaneously, which is what ERA does.

The clusters do not necessarily have to be homogeneous. For a example if we have eight 2-issue cores available we can form four 2-issue clusters or two 2-issue and one 4-issue or two 4-issue or one 8-issue (core/cluster). Figure 2.3 summarises that.

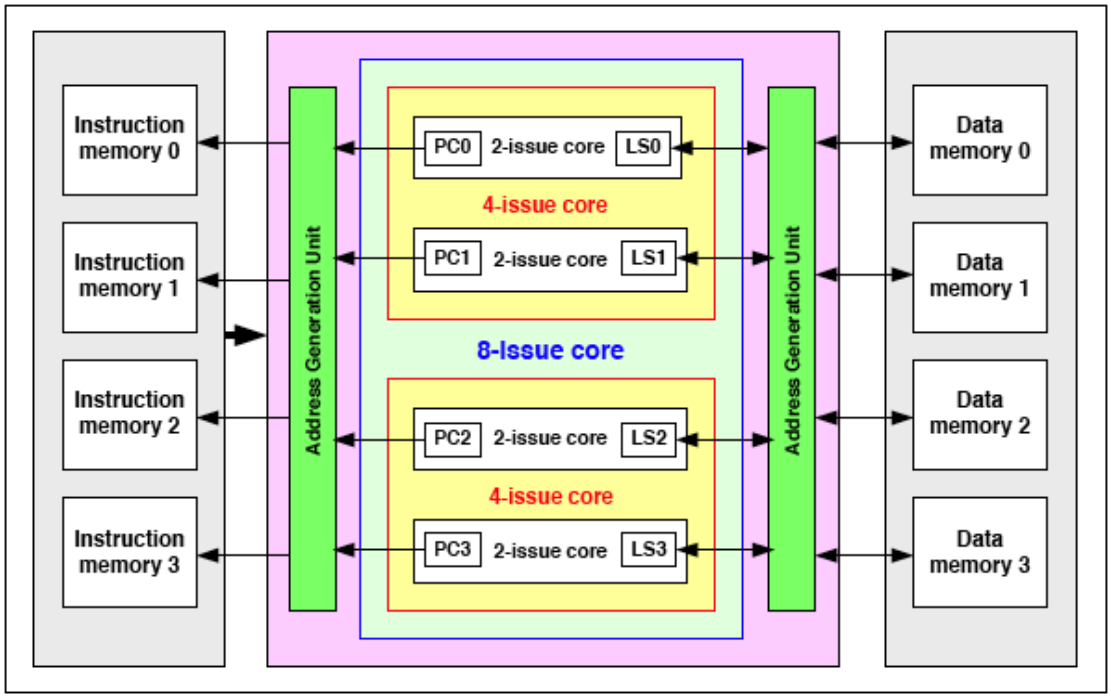


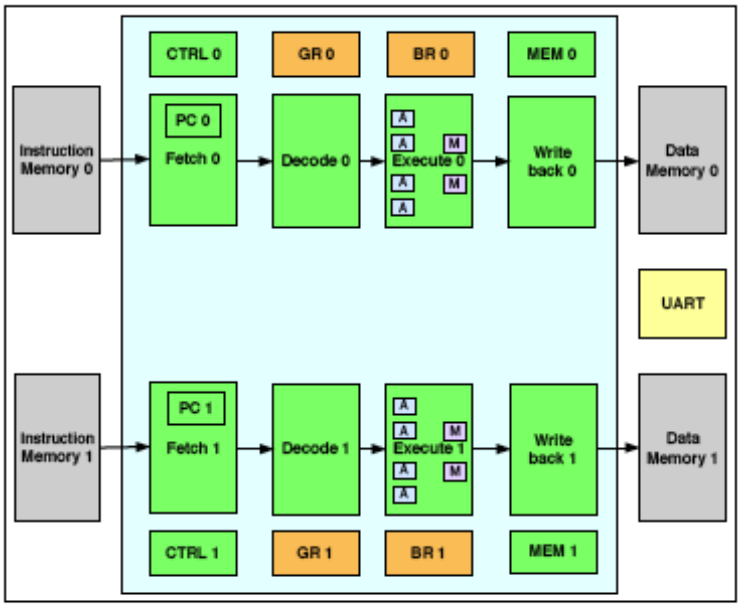
Figure 2.3: How ρ-VEX cores can be combined to form bigger ones.

It has to be noted here that not just any two cores can be combined. For example assuming that N=4 in Figure 2.1 and that every cluster is a 2-issue core, in order to form a 4-issue core we could combine cluster-0 and cluster-1 or cluster-2 with cluster-3, but not cluster-1 and cluster-2. To imagine all the possible combinations, thus, we can imagine the clusters as the leaves of a binary tree, where the 2nd and the 3rd leaf cannot be combined since they belong to different branches.

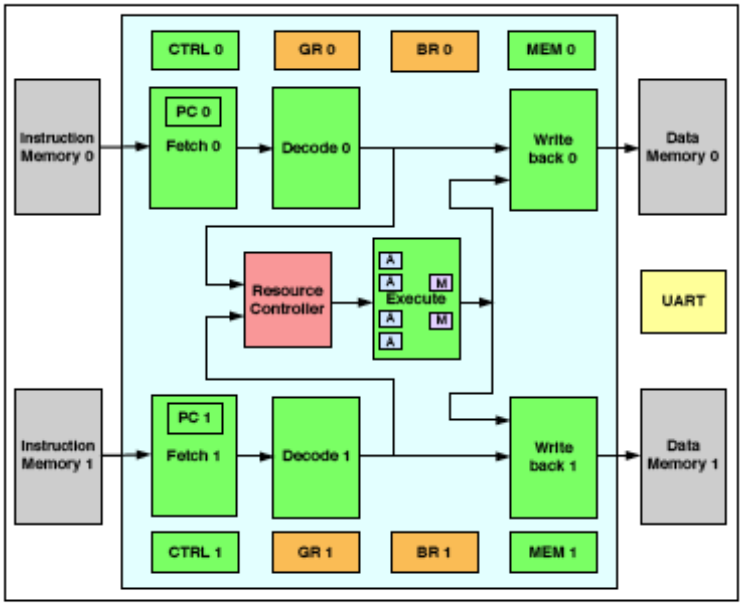
We owe this limitation to the fact that ERA applies a datapath sharing method by Anjam et al. [6] depicted in Figure 2.4. Without getting too much into the details of the method, we can mention that Anjam et al. manage to halve the resources needed for the execution units, by sharing resources. While one operation is writing back (WB) results to the data memory, another operation can make use of the execution units. Apart from the obvious benefits in resources, Anjam et al. report benefits in power consumption, an important element in Embedded Architectures. The limitation brings some performance drop for the Multicore Processor as we will show in Chapter 5, but that is not tragic and it gets compensated by first of the core speedup which [6] reports by getting rid of the inter-cluster communication delay and secondly the fact that this binary-tree topology quite simplifies the design of the hardware scheduler, which brings benefits both in terms of resources as well as clock speed.

Apart from the ability of the MCP to switch off cores to save energy [7], this design can allow the ERA MCP extend its fault tolerant support (such as on-the-fly switching on and off of fault-tolerance support to save energy [8]) by permanently deactivating

cores that give errors constantly and proceed with task scheduling by ignoring them and using only the rest of the computing resources.



(a) Non-shared datapath



(b) Shared datapath

Figure 2.4: Datapath sharing in ρ-VEX cores.

2.2 The VLIW approach

It has already been mentioned that ρ-VEX is a VLIW processor. A VLIW processor takes advantage of Instruction Level Parallelism (ILP)[9] to execute independent operations simultaneously next to each other utilising extra hardware that is available[10]. For example if we have two processing units, an adder and a multiplier, there is no reason to leave the adder idle, while the multiplier is busy. Thus we feed the adder (already during compilation) with another operation which has no dependency to the operation being executed in the multiplier.

The way superscalars do that, is to look further in the code to find independent operations. Most of the times they need to perform out-of-order execution of the operations to reach high performance. To implement that, as well as to make sure that the final result is equivalent to in-order execution, out-of-order superscalar processors make use of a lot of extra control logic. Not ideal for Embedded applications, where less hardware is not only cheaper and faster (also faster to build), but also less energy consuming. VLIW architectures move that workload to the compiler. The compiler looks for independent operations that can be executed simultaneously and packs them together into Very Long Instruction Words (VLIW). This way VLIW hardware is much simpler and does not have to support out-of-order execution.

The cost we pay with VLIW architectures is that not always is there enough instruction level parallelism to exploit. If we have a 4-issue VLIW processor for example, we need to feed it 4 operations on every single cycle. That is not always possible. So often we end up with No Operations (NOPs) in the code[11]. NOPs are zeros filling up the gap of the operations we could not find to make a full Very Long Instruction Word. Figure 2.5 depicts a NOP inside a 4-issue instruction word of a 16-bit operation width VLIW architecture. A NOP means that the execution unit that will receive it, will do nothing. Most VLIW research goes around the problem of minimising NOPs.

2.3 Parallel algorithms

ERA not only exploits ILP by using VLIW architecture, but in the same time the clustering system tries to exploit Data Level Parallelism (DLP). Some applications could benefit from splitting the data into blocks that can be processed in parallel and calling the task independently for each block. Although the latter is not the job of the hardware scheduler, making idle resources available for another programme or task sent to ρ-VEX by the operating system for execution is.

Parallel algorithms theory has a lot to offer for exploitation of intro-task parallelism, but not much when we are trying to map to hardware tasks totally independent from each other, as this is considered an NP-hard problem [17]. For example it offers useful

63	*48*	*31*	*15*	*0*
1101110111011101	1011010110110101	0000000000000000	1110101111101011	

Figure 2.5: A NOP in the 2nd position (bit-31 to bit-16) of a 4-issue VLIW.

tools for problems including communication (like Networks on Chip - NOCs), tasks with dependencies or communication with each other as well as divide and conquer methods for solving matrix problems [12], but nothing when it comes to completely independent tasks with no communication with each other competing for resources. Then the only fair solution in that case seems to be simply a FIFO and strict in-order execution.

However, there is a difference in ERA platform. The tasks can run in cores of several sizes, which creates some dependency between the tasks, cause some task B cannot use the resources allocated to some task A, till task A finishes. That would be a simple problem of balancing a tree graph, as painful as the word "simple" may sound in the the limited world of FPGAs and Embedded System. Figure 2.6 depicts such a weighted graph, where the nodes are the tasks and labeled by the name/number of the task and the size of the core (2-issue, 4-issue, etc) and the edge is the cycles it takes to execute this task in such a core. It is obvious that the bigger the core (thus the more resources we dedicate to this task) the shorter the edge (the faster the task is executed that is). On every level the sum of the core sizes is equal to the maximum resources we have. Let's say we have resources to build one 8-issue core or two 4-issue and so on as in the example of Section 2.1. Then 8 would be the sum of all the core sizes on each level (we let aside deactivating resources to save energy for now).

Since the maximum number of tasks we would be able to issue at such processor on one cycle is 4, we only take into consideration the next 4 tasks in the task queue. If all 4 can run simultaneously (Figure 2.6 left graph), obviously there is no need for balancing. The slowest task is giving us the height of the tree. If we dedicate double amount of resources though to the slowest task and let another task run only after this one or another one finishes, then we can shorten the tree's height. We managed to shorten the total execution of those tasks from 12 to 8 cycles. But then, what if we had made a different choice? Every task has 3 different versions (2-issue, 4-issue and 8-issue) and there are 4 different tasks. That means the task scheduler would have to construct $3^4 = 81$ possible trees (even though some of them are equivalent to others), calculate their heights and choose the shortest one. That would demand a whole Multicore Processor only for the scheduling, so it is out of option.

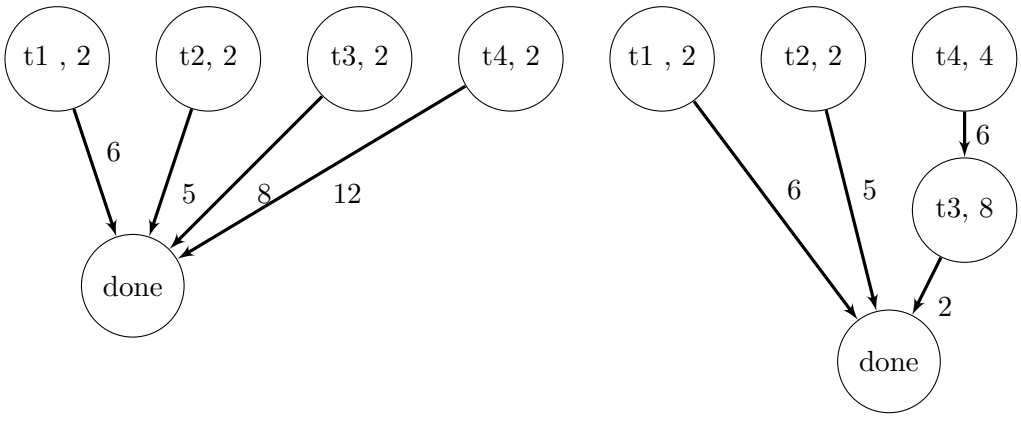


Figure 2.6: Two of the many scheduling possibilities of just 4 tasks in ERA.

Another option is to consider all nodes as 2-issue tasks and represent an 8-issue task as four 2-issue tasks with a line dependency. Then we we would end up with only one tree/graph like that in Figure 2.7 which has to be balanced. Apart from the fact that such a problem would still need a whole MCP on its own every time we would want to

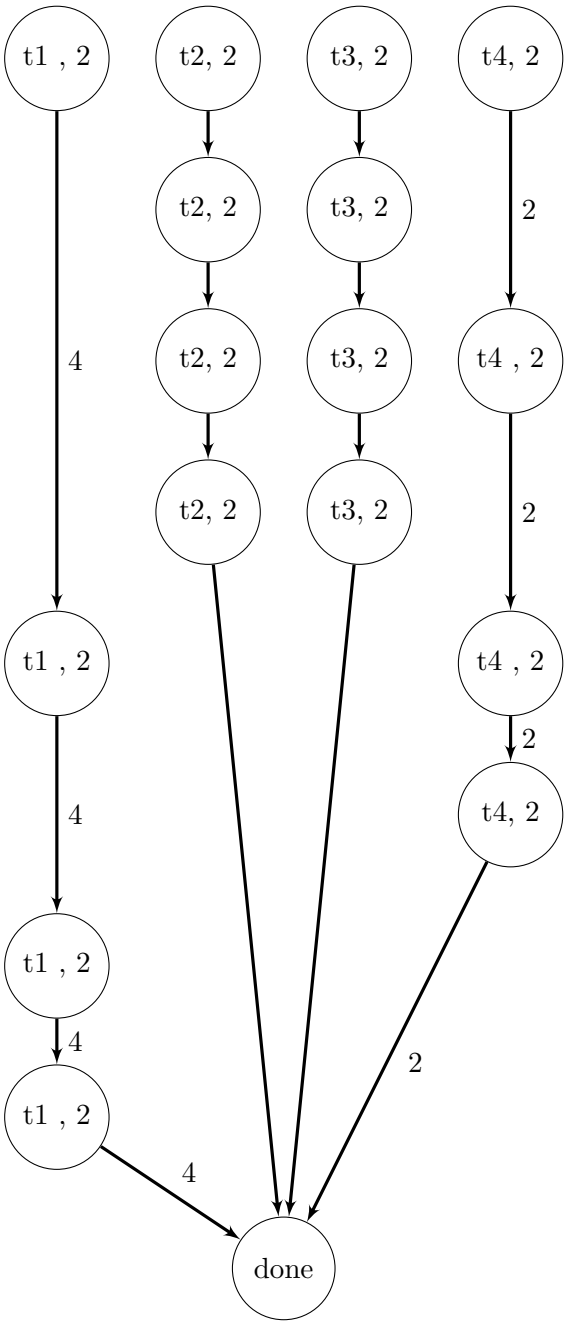


Figure 2.7: A scheduling graph with only 2-issue nodes.

reschedule the tasks, it is also not that accurate. In a perfect world the execution time of a task in a 4-issue core would be half the execution time of the same task in a 2-issue core. That means that we would manage to wrap all the operations of the 2-issue version into half the amount of instruction words we had in the 2-issue version without creating any new NOPs. Well, that is theoretically possible, but practically impossible, since no programme has such a great amount of instruction level parallelism. With other words pushing t1 up to execute it as a 4-issue task would not necessarily mean that the task would have been executed within 8 cycles.

Theory of Parallel algorithms and Parallel processing or Graph theory did not offer any solutions. However some investigation in the literature did take place, in case some algorithms have already been developed for similar to ERA platforms or conditions.

2.4 Related work

Natural research domains that can be related to the hardware scheduling of ERA are Grid Computers, heterogeneous multicore computers and homogeneous multicore computers. Each of which failed to supply us with an algorithm relevant to ERA; each for its own reasons. We present one typical research/case per cluster below.

2.4.1 Grid Computers

A Grid computer is a network of computers linked by software in such a way that they function as a single computer. The computational power of all those computers is combined to form a super computer[13]. The linking software has to perform task scheduling. The computational power of each computer in the grid is different, which could be related to the several ρ-VEX cores of different issue-widths inside the ERA multicore processor, but those computers are not next to each other, like the ρ-VEX cores, but they can even be at the other side of the planet. Thus most research on Grid task scheduling [14, 15, 16] , for example that of Keqin Li in [14], focuses on minimising the communication costs between the different computer elements. If the communication cost is not the only factor of those specialised task scheduling algorithms (for example in cases that all computers members of the gird are the same or quite similar) it is at least the most important one. But in case of ERA the communication cost is irrelevant, because it is the same, no matter what kind of core we are going to select.

2.4.2 Homogeneous Multicore Processors

Another cluster of research efforts is for scheduling on homogeneous multicore systems [17, 18, 19, 20]. Despite the fact that the geographical distances of the previous category of systems is not there, communication still is an important issue in Youness et al. [17], as the cores communicate with a shared memory through a bus as shown in Figure 2.8. Thus the algorithm is making a graph with dependencies (Figure 2.9) where round nodes is the execution time and hexagon nodes are communication costs. The algorithm is worrying mostly about queuing up tasks with dependencies after each other on the same core, in order for the next task to find the data ready in the core through

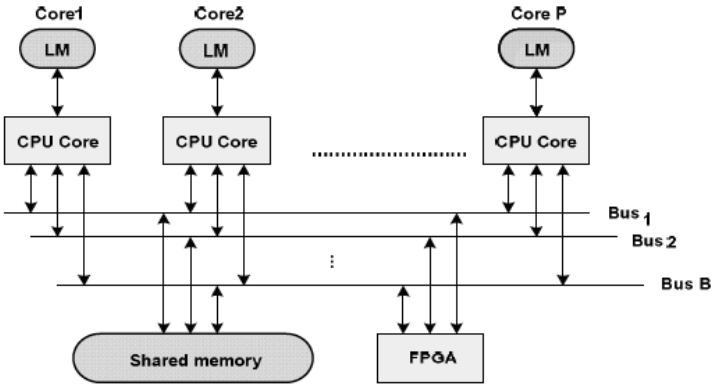


Figure 2.8: Homogeneous multicore system

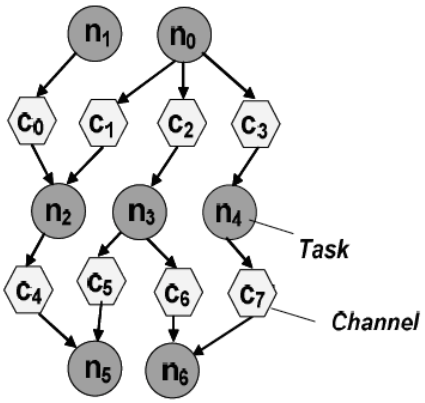


Figure 2.9: Scheduling graph for a homogeneous multicore system

port-forwarding or core cache instead of having to use the bus to access the main memory. Such a scheme does not match the specifications of ERA, since we assume that tasks are totally independent, but could be used later as an improvement. For now the ability to run a task on a bigger core than the others and wether a scheduling that takes advantage of that is possible is a much more interesting question and most likely not to be answered by research efforts concerning homogeneous systems. Even in publications where tasks are independent (as in [20], since the system is homogeneous, the algorithms target mainly on workload balance which is core-oriented rather than task-oriented.

2.4.3 Heterogeneous Multicore Processors and Multiprocessor Computers

With the exception of some moments that only cores of the same size are used, ERA MCP can be considered a heterogeneous multicore system. Scheduling in heterogeneous has also not been very impressive[21, 22, 23, 24, 25], most authors claiming simply effective or efficient scheduling, but none of them optimal. The most interesting case is

that of Tang et al. [21]. The authors use estimated execution time of the task for every core in order to decide on which core the task will be assigned, but important role on the decision takes the deadline, since this is research for, not just Embedded Systems, but Real-Time Embedded Systems. Deadline miss rate is also the only criterium on which they compare the different versions of their algorithm. Moreover it is a software scheduler, although not that hard to implement in hardware. This could be a future add-on for the ERA hardware scheduler in order to support real-time systems, but it cannot be used as the basic scheduling algorithm.

2.5 Conclusion of Chapter 2

After getting an inside look of the Processing component of the ERA platform and understanding the particularities of it, as well as of its main building block, the ρ-VEX core, we tried to find in the bibliography an available scheduling algorithm to implement as the Hardware scheduler of ERA. This literature research did not yield any results, both because of the complexity of the problem, as well as the pioneer characteristics of ERA that we would like to take advantage of. Thus, we will try to design simple scheduling algorithms, especially tailored for the ERA platform and see how they perform.

3 Implementation

From the Introduction (Chapter 1) and the flow chart of Figure 1.2 the reader should probably have imagined already that the evolution of this research would be nothing like linear, which makes it rather difficult to lay on a book. However, this chapter and the next one will attempt to present the research in a linear way.

We saw in the previous chapter that the relevant literature could not supply us with the appropriate algorithm for the Hardware scheduler of the ERA platform. Thus, we went on and designed 4 simple scheduling algorithms. Some of them were part of the initial planning, whereas others came up or were abandoned during the course of the research. In this chapter we present those algorithms, their characteristics and their differences.

Imagine that our ERA Multicore Processor of Figure 2.1 is a co-processor in an embedded system and the main processor sends to the MCP only the hardest tasks for execution. The Hardware scheduler receives a task in a task list (FIFO) and tries to decide what percentage of the resources it should allocate to this task.

The Monitoring hardware (Figure 1.1) supplies the Hardware scheduler with information about the task. Information on how fast it has run in the past on every core-size available, so that the Hardware scheduler can make a smarter decision. Smarter meaning we do not have to offer an 8-issue core to a task if the execution-time gain compared to this task run in a 4-issue core is just 5% or 10%. That means that there is not much of Instruction Level Parallelism (ILP) to exploit in this task and that the 8-issue version is full of NOPs. This information can also come from the Operating System (OS) accompanying each task in order to save on hardware resources.

3.1 A basic scheduling algorithm

3.1.1 The naive approach

Assuming we have a MCP of corespace 8. That means that we have resources to implement one 8-issue core or two 4-issue cores and so on, just like in the example of Section 2.1. There are 2 tasks running, one on a 4-issue core and one on a 2-issue core. That means that our only free resources is a 2-issue core. The situation is depicted in Figure 3.1 below.

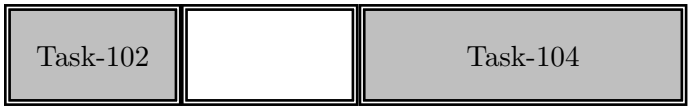


Figure 3.1: A 2-issue and a 4-issue task running in an ERA MCP.

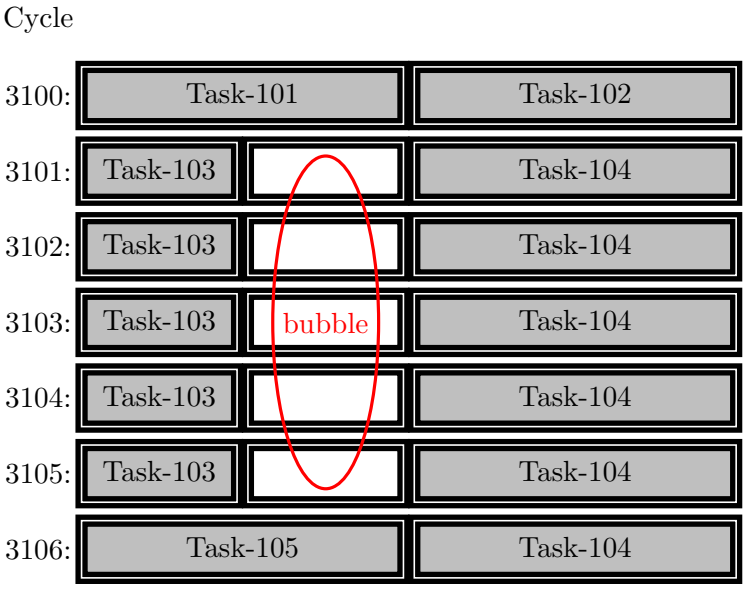


Figure 3.2: A bubble created in the execution timeline.

If the first task in the task list, waiting for execution, comes with the indication of preferably running on a 2-issue core, we got lucky. But if the task would prefer to run on a 4-issue or 8-issue core, then the Hardware scheduler has a decision to make. The simplest thing the Hardware scheduler can do is to wait until resources become available. But that means a NOP would have to be added in the 2nd cluster till one of the two tasks is finished. That creates a bubble in the execution timeline as seen in Figure 3.2. Such bubbles cause the utilisation percentage to drop, increase the task latency, which is the time from the moment a task arrives at the task list till the time its execution is completed, and the total execution time for any chosen amount of tasks.

In order to minimize the number of the bubbles, we can allow the algorithm to look further in the task list for smaller tasks that can be executed inside those bubbles. As in hardware the resources are not infinite there should be a maximum window in the task list that the algorithm can search for smaller tasks. This is the simplest algorithm we can implement, it is called 'Basic' (or 'Algorithm A' or just 'AlgA' for short), and it is depicted in Algorithm-figure 1 below.

Algorithm 1 Basic or Algorithm A

```
1: for i = 1 → 10 do
2:     find space p where task(i) fits
3:     if p ≠ NULL then
4:         allocate p to task(i)
5:     end if
6: end for
```

3.1.2 Time complexity analysis

With the first look, one would estimate the time complexity of this algorithm to be $O(m \times n)$, m being the window size and n being the maximum number of 2-issue cores that the platform can offer to the tasks, or half the 'corespace', which can be defined as the maximum ops that can be running on a specific ERA MCP at the same time. The hardware world however has limitations and advantages that work on the benefit of our algorithms. First of all, we do not expect the window to be too large because, as we know from superscalars, that would seriously increase the complexity of the hardware design. Even if the window-size is adaptable on-the-fly (instead of decided on design-time) it would never exceed 10. That drops the time complexity to $O(n)$. In the world of software it is difficult to overcome the $O(n)$ time complexity, which comes from the search for an available core of the size that task(i) optimally runs (unless we use parallel computing which still has its limitations), but in the world of hardware we can turn the time complexity $O(n)$ into workload complexity $W(n)$ or any other $W()$. By means of logic gate design we can come up with a circuit of OR gates, like the one in Figure 3.3 that manages to detect a suitable free core within 1 cycle, thus time complexity $O(1)$.

3.1.3 A realistic basic algorithm

As we will see in Chapter 5, the Basic algorithm has a very important disadvantage that makes it unusable. The algorithm tends to execute all 2-issue tasks in the list first, then the 4-issue tasks and finally the 8-issue tasks. Even if we are not talking about real-time systems, where tasks have deadlines, this is still a problem, because in the real world that tasks keep coming constantly, it means that bigger tasks will never be executed. They will be waiting forever for execution because 2-issue tasks will be

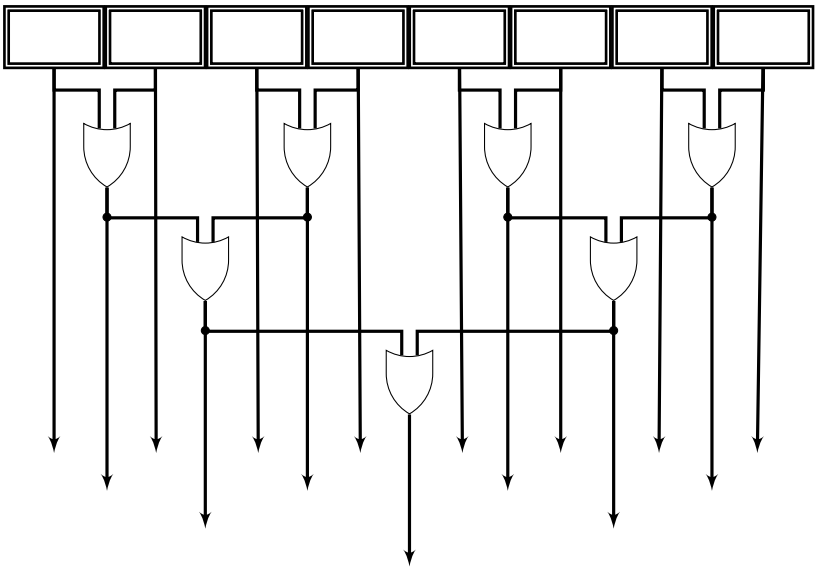


Figure 3.3: The logic that detects available cores of any size (2,4,8,16) on an ERA MCP of corespace 16. Output Low means available, whereas output High means unavailable.

coming just on time to grab small cores immediately as soon as they become available. For this reason a modification is necessary. One modification could be that we first execute all tasks within the window-buffer before we start filling it up again with new coming tasks. But that would cause bubbles again and thus the utilisation would drop. Another similar modification is to introduce barriers/checkpoints wherever we meet an 8-issue task. That means that as soon as the algorithm detects an 8-issue task it will stop feeding the window-buffer with new tasks, until it is empty. Until that 8-issue task has been executed, that is. The 4-issue tasks are much more frequent than 8-issue ones. So the window-buffer often gets full of them and eventually it is forced to run them. That is sufficient so that they do not get postponed forever and works almost like the barrier function. Introducing a barrier-function also for 4-issue tasks would create too many bubbles. We call this version 'Realistic Basic Algorithm' or 'Realistic Algorithm A' or 'RAlgA' for short.

3.2 Versioning

Some 4-issue (and above) tasks though are not that large, meaning, they do not take that long to execute. Making them wait for several hundreds of thousands of cycles for free resources, while they take several hundreds of cycles to execute, even in the smallest cores, would be criminal. For those tasks it might be smarter to do the effort to bring another version of the task, compiled to run on a smaller core and allocate to them on such a smaller core which is available immediately. This algorithm is called 'Versioning'.

3.2.1 Communication Penalty

We will suffer a communication penalty in that case, however, since we do not expect the Instruction Memory (cache) to be big enough to hold 3 versions of every task in the task list. Thus communication with the main memory is expected and that will bring penalty to the execution time of the task. We hope this way that the overload from the slower execution plus the penalty will be in total less than the time latency of the task if it would be waiting for free resources as is the case with RAlgA. The size of the penalty depends on the design of the platform, thus will be a subject of research.

An example is shown in Figure 3.4. Task 101, a 2-issue task is already running on this MCP of corespace 4, when the 4-issue Task-102 arrives. AlgA would make Task-102 wait until Task-101 is done and thus enough resources are available. That would create a bubble of 5 cycles and Task-102 would only start running at cycle 3106. RAlgA might have done the same, unless it could find another 2-issue task to cover the bubble, but if that task took more than 5 cycles to execute, that would be even worse for Task-102 because that would mean it would have to wait even more to start running. Assuming that Task-102 takes 2 cycles to execute in a 4-issue core and 4 cycles in a 2-issue core and that the penalty to bring the 2-issue version is 2 cycles, we see that the task would finish 1 cycle earlier in Versioning, letting Task-103 also start earlier. Although 2 cycles

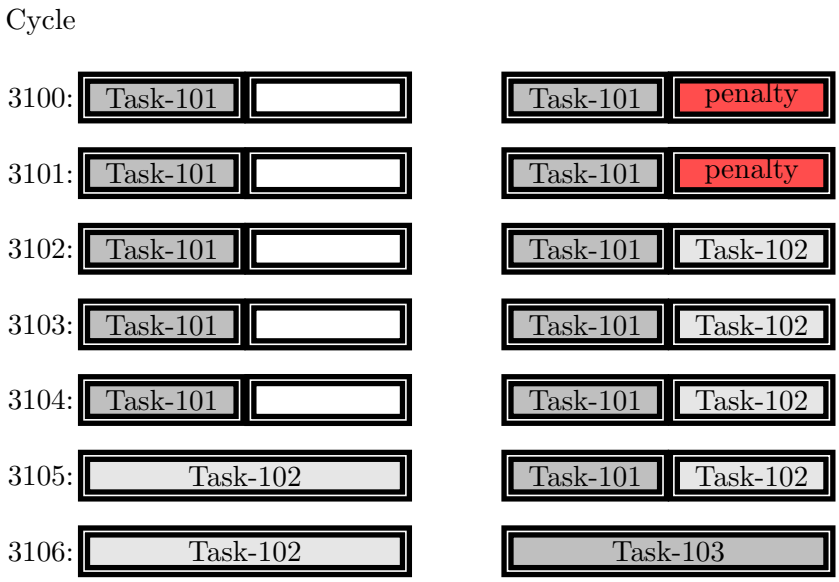


Figure 3.4: A scenario on AlgA (left) and on Versioning(right).

reload penalty looks quite a heavy cost for a 2-cycle task, plus 2 cycles extra executional penalty (since 2-issue cores are slower than 4-issue cores) a total of 4 cycles penalty, the execution time of the 4-issue task is not what the Versioning penalty is competing with, but the task latency in the Basic Algorithm. Thus, as long as the following is true

$$T(i, rest) + T(j, 2 \times s) > P(j, s) + T(j, s) \tag{3.1}$$

(T being the execution time and P being the reload penalty, i being a task already running (like Task-101 in Figure 3.4) and j being a new task coming (like Task-102 in the same figure) and s being a core size) Versioning is better than the Basic algorithm.

Algorithm 2 Versioning

```
1: for s = task(i).size → 2 do
2:     find space p where task(i) fits
3:     if p ≠ NULL then
4:         if s ≠ task(i).size then
5:             bring the right version of the task
6:         end if
7:         allocate p to task(i)
8:     end if
9: end for
```

Trying to include (in)equation 3.1 in the Versioning algorithm would demand extra resources (especially if we want to implement it within 1 cycle we would have to calculate the $T(i, rest)$ for every i, for every task running at the moment that is) and the benefit would not be certain, since how long a task takes to run is not only dependent on the size of the core it is running on, but also on the amount of data it has to process. The algorithm would obviously fail if it let the new task wait, thinking a task would soon finish, while the task running actually has to process this time double or triple the amount of date it had to process last time.

Thus Versioning shall *always* schedule a task on every cycle (by reloading its binary to a version that fits) provided that there is an idle core. That means that the hardware utilization will be 100%, contrary to AlgA and RAlgA which produce task bubbles.

3.2.2 Time Complexity

The Versioning algorithm (seen in Algorithm-figure 2) has the same time complexity as Basic, $O(1)$. The number of the iterations of *For* cannot exceed $logm$, where m is the size of the biggest core which that specific ERA design supports. If, for example, the decision taken on design time is that the maximum core size is 8, then that *For* will have to run once for $s = 8$, once for $s = 4$ and once for $s = 2$. A time complexity of $O(logm)$. Taking advantage of the logic circuit of Figure 3.3 (the output of which can be controlling some multiplexers as shown in Figure 3.5) we can even perform all three iterations in 1 cycle. That means the hardware version of the algorithm will have $O(1)$ time complexity.

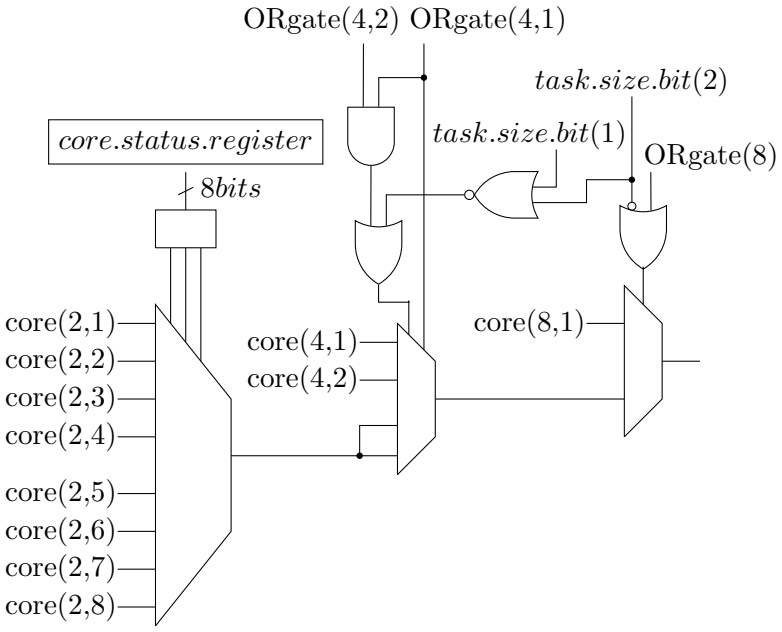


Figure 3.5: Parallelising downgrading.

task	GB		Ver./Basic		slowdown	
	2-issue	4-issue	2-issue	4-issue	2-issue	4-issue
1001	29.750	20.229	31.546	18.376	-6%	10%
1002	1.003	614	807	516	24%	19%
1003	11.202	10.337	10.488	10.335	7%	0%
1004	143.178	105.316	128.600	98.526	11%	7%
1005	14.054	10.056	11.037	9.208	27%	9%
1006	80.272	50.562	73.550	45.248	9%	12%
1007	798.108	626.721	671.589	572.852	19%	9%
1008	640.728	481.976	552.000	448.218	16%	8%
1009	788.241	538.547	562.866	469.166	40%	15%
1010	1.897.159	1.339.742	1.599.351	1.220.330	19%	10%
1011	34.910	22.605	24.845	18.018	41%	25%
1012	39.644	29.880	35.334	27.781	12%	8%
1013	3.077.053	2.285.802	2.310.467	2.026.995	33%	13%

Table 3.1: Execution times (in cycles) and GB slowdown to Versioning/Basic.

3.3 The Generic Binary

A great innovation would be if we could change to another core size than the one that task prefers, either from the beginning or in the middle of the execution of a task, without paying that penalty. Brandon and Wong [26] achieved to compile any task to a binary that can run on any core, even migrate to another core on-the-fly with a mere penalty of 155 cycles[27] and still run correctly. It is called generic binary and it makes the ERA ρ-VEX the first reconfigurable VLIW processor that can do that[28]. The algorithm takes the name from that special binary too, so for the rest of the thesis the term 'Generic Binary' (GB) will refer to the algorithm, not the binary.

3.3.1 Benefits and disadvantages

The penalty of 155 cycles is not the only one, though. A task compiled for the Generic Binary algorithm will not run as fast as if it were compiled specifically for a 2-issue or a 4-issue core (8-issue cores have no difference compiled to GB or not) for use in Versioning and AlgA. A task compiled for GB will usually take about 10% to 40% more time to execute than the same task specifically compiled for 2-issue or 4-issue (Table 3.1), depending on the task. However being able to downgrade or upgrade will give us the opportunity to support priority tasks, interrupts and save on energy, because downgrading with GB consumes less energy than reloading with Versioning and generally running a task on a smaller core also saves energy (assuming no other task comes to use the resources the downgraded task left idle)[28].

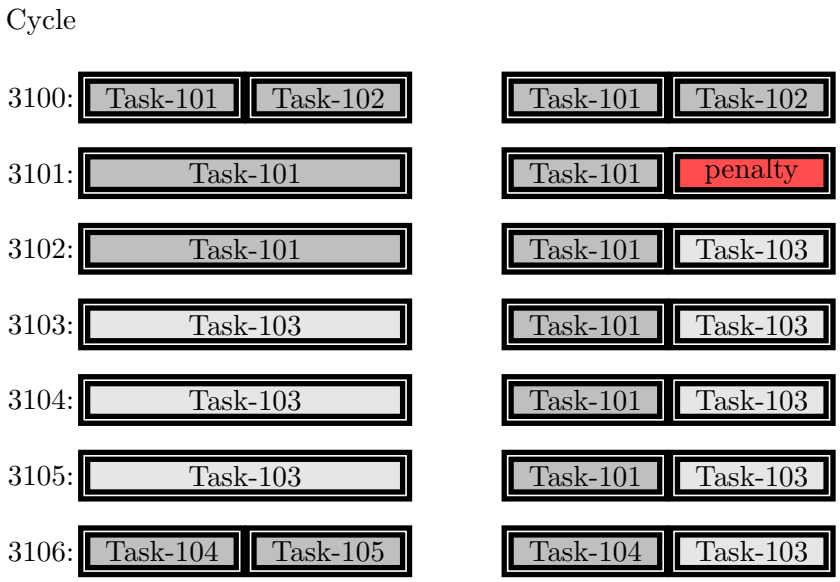


Figure 3.6: GB (left) and Versioning (right) handling a new 4-issue task.

3.3.2 Priority tasks

Priority tasks are tasks that:

- either have a deadline, so they need to use more resources to finish on time(Figure 3.6),
- or tasks that just came in and need to run now (Figure 3.7 and 3.8), even if it appears there are no available resources
- or that tasks that the operating system for any reason decided that they are more important than the rest(Figure 3.9).

Versioning cannot support those tasks. Switching to a bigger core to catch a deadline or downgrade to a smaller core to let a priority task run would mean that we would have to start running the task all the way from the beginning again. The only thing Versioning could do and only for tasks that come in and have to run now (not tasks that are already running and suddenly they acquire high priority) is freeze a (non-priority) task that is running, store its context and let the priority task run in its place. As soon as it is finished, the old task can be loaded again and start from where it had stopped.

GB on the other hand can handle all three kind of tasks described above. Tasks with a deadline can be given priority, either they are running now or they just came in. If the operating system (or the Hardware monitor) is afraid a task can lose a deadline, it can upgrade it by letting the task take over adjacent resources instead of bringing in a new task. At this moment the algorithm only looks for available recourses in adjacent resources. But total relocation of the resources on every cycle is feasible with some modifications. That means that the algorithm will switch location of two tasks, say A

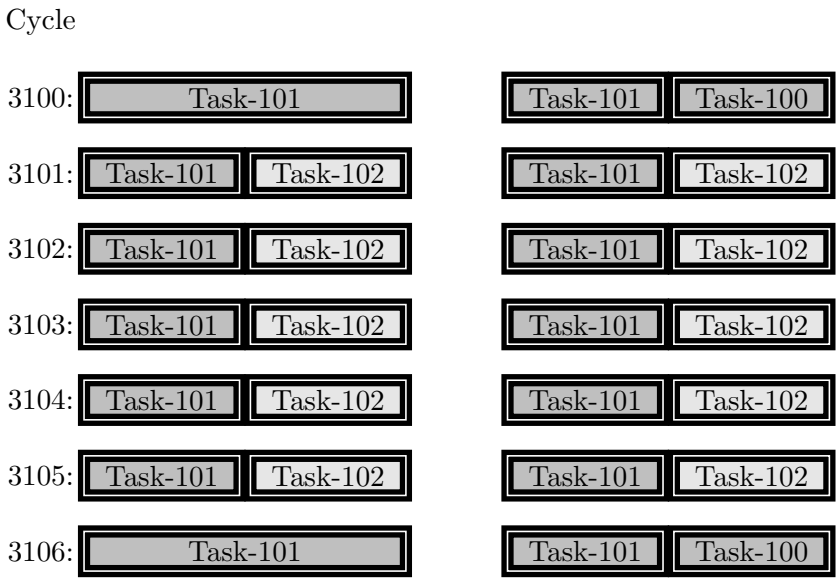


Figure 3.7: GB stealing resources from a task(left) or freezing it (right).

and B, if it sees task A can take advantage also of adjacent idle resources of task B, if task B cannot or simply would really benefit as much.

When a priority task comes in and no resources are available the algorithm can either steal resources from another task. That could mean that the other task either has to downgrade or that it should or freeze and be taken out of the MCP to give its resources to the priority task. If victim-task is a non-priority task the choice is obvious. If it is also a priority task the question what to do rises. For the time being the algorithm supports only 1 level of priority. But it can easily be modified to support multiple levels of priority. The task with the smallest priority obviously becomes the victim-task.

There is a figure exhibiting every one of those cases. In Figure 3.6 Task-101 and Task-102 are running. Task-102 is done and the next task is Task-103, a task that preferably runs on a 4-issue core. GB also notices that Task-101 is a 4-issue task which has been downgraded. It upgrades to run in 4-issue so that it runs the 4 remaining words (8 operations) in 2 cycles instead of bringing Task-103. Then there is enough space for Task-103 to run on a 4-issue core, no need to downgrade. This way Task-103 finishes within 3 cycles instead of 6. Versioning followed a different pattern. It cannot upgrade Task-101, since it would have to make it run all the way from the beginning. So it has to downgrade Task-103. Task-103 runs 15% faster on Versioning since it is compiled especially for 2-issue, but it has to pay the re-loading penalty of at least 1 cycle. Both tasks finish later in Versioning than in GB. Even in the case that the GB overload for Task-103 was higher and thus Task-103 would also end on cycle 3105, there would still be the benefit of Task-101 finishing earlier.

In the case that a new priority comes in and there is no place for it to run is depicted in Figure 3.7. In the left scenario Task-101 is non-priority task (or a task with lower priority) running already on a 4 issue core. Task-102 is a (high) priority task that comes in and needs to run immediately. In that case Task-101 will be downgraded to a 2-issue

task	GB					Versioning/Basic			
	execution times			gain		execution times		gain	
	2-issue	4-issue	8-issue	2→4	4→8	2-issue	4-issue	2→4	4→8
adpcm	29750	20229	17025	32%	16%	31546	18376	42%	7%
bcnt	1003	614	515	39%	16%	807	516	36%	0%
blit	11202	10337	10334	8%	0%	10488	10335	1%	0%
compr.	143178	105316	95697	26%	9%	128600	98526	23%	3%
crc	14054	10056	9209	28%	8%	11037	9208	17%	0%
des	80272	50562	37729	37%	25%	73550	45248	38%	17%
engine	798108	626721	583252	21%	7%	671589	572852	15%	-2%
fir	640728	481976	421147	25%	13%	552000	448218	19%	6%
g3fax	788241	538547	464712	32%	14%	562866	469166	17%	1%
jpeg	1897159	1339742	1130842	29%	16%	1599351	1220330	24%	7%
pocsag	34910	22605	17024	35%	25%	24845	18018	27%	6%
qurt	39644	29880	26053	21%	25%	35334	27781	13%	6%
v42	3077053	2285802	2006688	12%	26%	2310467	2026995	12%	1%

Table 3.2: Execution-time gain from upgrading (8-issue times common for all algorithms)

core to leave some resources available for Task-102. If Task-101 still running when Task-102 finishes it can be upgraded again to a 4-issue core. On the right side, there are two tasks running when (high) priority Task-102 comes in, specifically the (low) priority Task-101 and the non-priority Task-100. GB will stall Task-100, store its context to the memory and let Task-102 run. When Task-102 is done, Task-100 can come back for execution. The scenario on the right side is what the other algorithms would do too, even if there was a 4-issue task running that moment. The benefit of GB is that it lets Task-101 keep running while the priority task is also running. That could be compensated in the other algorithms by letting the priority task use as many resources as possible. But not all tasks benefit significantly from bigger cores as we can see in Table 3.2. The execution times as those of Table 3.1 are supplied by [26].

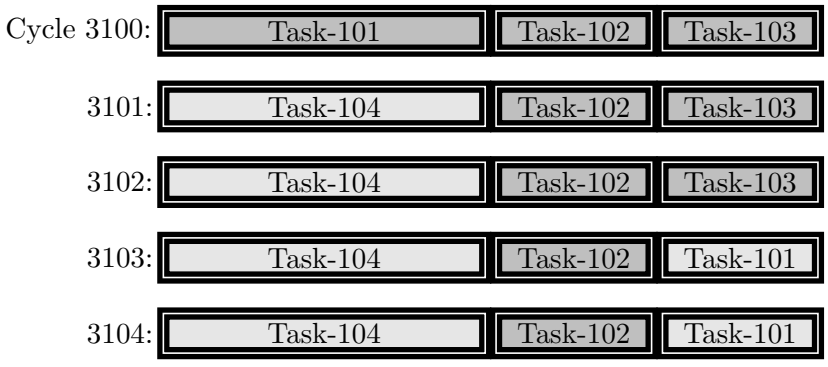


Figure 3.8: A 4-issue forced priority task in an ERA MCP of corespace 8.

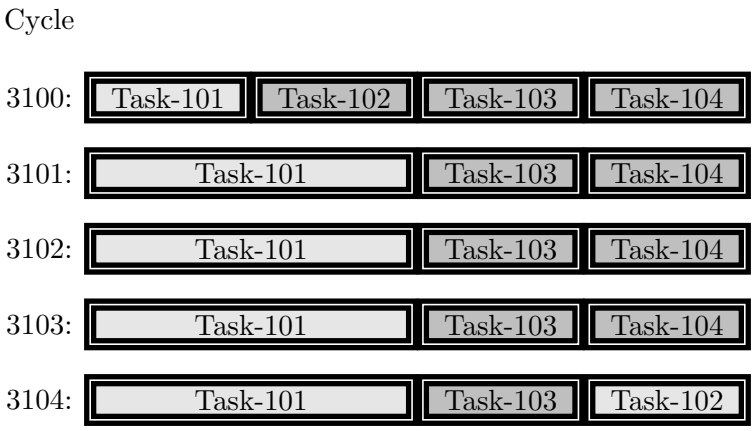


Figure 3.9: A task suddenly gains priority.

Figure 3.8 presents the case of a forced priority task. That is a task that not only has to run now, but it also has to run in the preferred core size, by all means. In that case Task-101 is taken out of the processor and comes back after resources are available. In this scenario one of the smaller tasks happens to finish before the forced priority task, so Task-101 is downgraded and brought back to a smaller core instead of keep waiting for the forced priority Task-104 to finish.

And finally Figure 3.9 depicts the case of a task that suddenly required priority and needs to take over resources around it. In this case this is Task-101 and Task-102 has to be taken out of execution. If another task is done in the meantime, like in this case Task-104, then Task-102 can come back to execution. Otherwise it has to wait for Task-101 to finish.

3.3.3 Interrupts

A big advantage of GB is that it can handle interrupts without having to freeze all the tasks running. Given that there is at least one 4-issue or higher task running at the moment, we can downgrade it and let the interrupt be handled in the core that will be set free. Interrupts are identical with priority tasks of the cases depicted in Figure 3.7.

3.3.4 Time complexity

GB can be represented in algorithmic form as seen in Algorithm-figure 3 below:

Algorithm 3 Generic Binary or Algorithm B

1: **for** $s = task(i).size \rightarrow 2$ **do**
2: find space p where $task(i)$ fits
3: **if** $p \neq NULL$ **then**
4: allocate p to $task(i)$
5: **end if**
6: **end for**

It looks exactly like versioning, but the reloading logic is missing. Thus, $O(1)$ complexity also for GB. Priorities could be implemented in parallel, in a similar way as the logic in Figure 3.3 and 3.5 but using the information from the priority bit instead of the allocated bit of the core-status register and using the size-check only if the task has a forced priority. With a priority task we do not care if there is another task running on the core or not, but only if it has a priority. After it is decided on which core the priority task is going to run the core-status bit simply tells us if we have to freeze or downgrade the running task before we bring the priority task.

3.3.5 GB++

Generic Binary with default upgrading can be implemented by adding a preliminary step before the *for* as showed in Algorithm-figure 4 below:

Algorithm 4 Generic Binary++ or GB++

find task t such that $p(t).size < task(i)size \&\& (p(t-1) = NULL || p(t+1) = NULL)$
if $t = NULL$ **then**
    for $s = task(i).size \rightarrow 2$ **do**
        find space p where $task(i)$ fits
        if $p \neq NULL$ **then**
            allocate p to $task(i)$
        end if
    end for
else
    upgrade
end if

To sustain O(1) complexity we have to add logic to the circuits of Figure 3.3 and 3.5 but that would be only a couple of AND/OR gates of this kind:

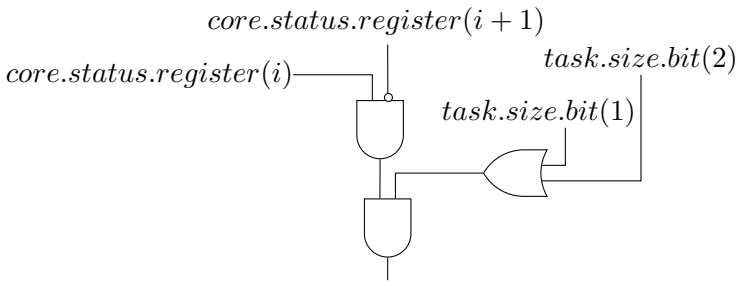


Figure 3.10: Logic for parallelising default upgrading of core i.

This would be the logic per implementable core, which means also cores of 4-issue and 8 issue. The only difference for the bigger cores is that the $core.status.register(i+1)$ in Figure 3.10 is replaced by the outputs of the ORgates of Figure 3.3. Thus for corespace 8 we would need $8 + 2 + 1 = 11$ times the circuit above. We pay $W(n)$ complexity to preserve $O(1)$ time complexity.

3.4 AlgD and AlgBall8

If one looks at the scheduling historic of GB they will notice that the algorithm is downgrading too much. Sooner or later 2-issue execution dominates and the chance that two 2-issue tasks finish in the same time so that enough resources for a 4-issue task are available is rather small. In an effort of preventing too much downgrading we tried to switch the preferred core of all tasks to the biggest possible core (for this research that is 8) in AlgB/GB ending up with 'AlgBall8'. This is not really exactly a new algorithm, it is still AlgB/GB, but the input is altered so that it appears that the preferred core of every task is 8-issue. The results were not better as we will see in Chapter 5.

Another way of avoiding too much downgrading was to combine GB and AlgA, creating AlgD. This algorithm downgrades the tasks only one level down. If there is still no space to run the task it looks for another task in the window and tries again later on. Unfortunately this algorithm showed worse results than GB, probably because it combines the disadvantages of both algorithms, with most important the disadvantage of AlgA to create bubbles.

3.5 Conclusion of Chapter 3

In this chapter we presented the several algorithms that were developed and tested during this thesis, starting from a very Basic task scheduling algorithm, moving to algorithms more specific to the ERA platform. Such as Versioning that downgrades the task by scheduling it to run on a smaller core to avoid creating bubbles that Basic does. Running in a smaller task means bringing from the memory a different binary of the task, compiled especially for that core. That takes time, which is called the 'reloading penalty'. To deal with the reloading penalty Generic Binary was invented creating a binary of a task that can run on any core. That also created the ability to downgrade tasks on-the-fly to make space for new tasks, support priorities and handle interrupts without big delays for the tasks already running. All these lead to a new algorithm, AlgB or GB. But it also brought the idea of upgrading a task when resources become available, instead of bringing immediately a new task for execution. That is GB++. Finally some very unsuccessful algorithms like AlgBall8 and AlgD were mentioned. In Table 3.3 there is an overview of all the algorithms that were developed, their characteristics and their differences. It is time now to see how those algorithms performed and how they compare to each other.

character. /situation	**algorithms**					
	AlgA	**RAlgA**	**GB**	**GB++**	**Versioning**	**AlgD**
task does not fit	stall task	stall task unless 8-issue	downgrade task till it fits	downgrade task till it fits	downgrade task till it fits	downgrade once then stall if still does not fit
window	yes	yes	no	no	no	yes
downgrad.	no	no	yes	yes	only before execution	only once
upgrading	no	no	no	yes	no	no
penalty	no	no	practically not (155 cycles)	practically not (155 cycles)	yes	practically not (155 cycles)
interrupts	replacing bubbles or stalling	replacing bubbles or stalling	downgrading or stalling	downgrading or stalling		12%
priorities	only static	only static	only static	all priorities	only static	no
forced priorities	only by stalling	only by stalling	yes (without upgrading)	yes	only by stalling	no
binaries per task	3	3	1	1	3	1

Table 3.3: Overview of the developed algorithms.

4 Simulator and benchmarks

4.1 The simulator

Although there is a simulator available for the ERA platform testing, it does not support yet multitasking. That means the tasks have to run the one after another. Upgrading and downgrading and interrupt support is already implemented, but still only when one task is running. Going immediately in the simulator and implementing task scheduling and all the algorithms above would add non-scientific overload to this thesis and it would make every experiment slower.

Since we already have the average execution times of several benchmark tasks for every core size available, it is a better idea for now to design a small, light and fast simulator only for the task scheduling. The simulator was implemented in C, went through 18 different versions during the development of all the algorithms we described in Chapter 3, each version consisting of more than 2.000 lines of code.

The simulator in its final version has several tools as shown in Figure 4.1. Such tools are the Task list generator, the Priority scenario generator, the Algorithm simulator itself, as well as some small scripts that test the outputs of all the previous tools or give statistics about them.

4.1.1 Task list generator

The Task list generator is where every experiment starts from. It takes as input a file that includes the average execution times of the benchmarks for every type of core and their preferred core, as well as the desired size of the task list as number of tasks. It gives as an output a task list of the desired size made of the tasks of the input file randomly placed all over the list.

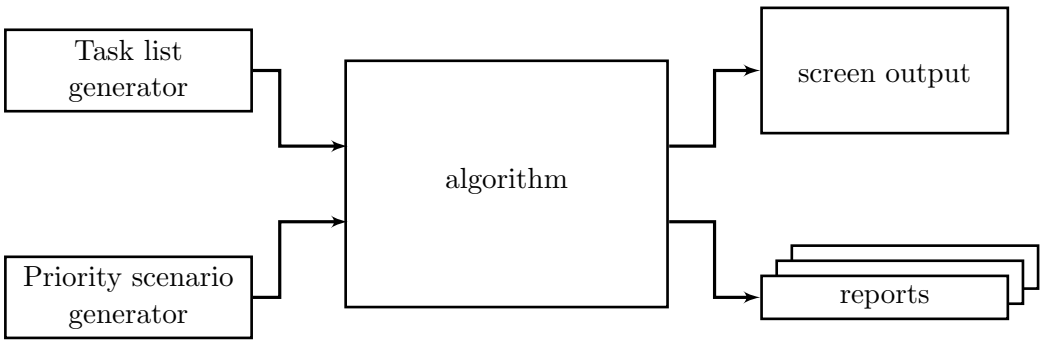


Figure 4.1: An abstract diagramme of the simulator.

This is a typical line of a task list:

1002 4 807 516 515

The first number (1002) is the task-ID. The second number is the preferred core. This one is derived from rest three numbers which are the typical/average execution times of that task in (from left to right) 2-issue core, a 4-issue core and an 8-issue core (a rather small taks in this case). The preferred core could be calculated just on the cycle that we schedule the task. However, it is very handy to have it stored already (especially when we try to see if a task is upgradable or checking priorities) and update it only after a task has finished running. This could happen with dedicated hardware or with software/interrupt.

4.1.2 Priority scenario generator

The priority scenario generator produces a random list of priorities. Its input parametres are the name of the scenario, the number of tasks in the task list it is going to be used with and the density of the priorities as a percentage. It then produces an output of which the first line is the name of the scenario and the priority density (as a percentage) and then a list zeros and ones, zero for a non-priority task and one for a priority task. The Algorithm simulator couples those priorities with the task list generated by the Task list generation. The length of the two lists should be equal. This the beginning of a typical output of the Priority scenario generator:

45 10
0
1
0
0
0
...

It means that this is scenario 45 with 10% priority density and that the first and the third task are non-priority tasks whereas the second is a priority task.

4.1.3 Algorithm simulator

The Algorithm simulator is the biggest part of the code. It takes as input the lists the other two generators produce (the priority list is not compulsory) and simulates the execution of the task list. Other input parametres are whether priorities/interrupts should be used or not, whether the user wants output on the screen apart from the report and the algorithm with which the task scheduling should be performed. The rest of the input parametres depend on the algorithm. For AlgA and RalgA the user has to give the window size, for Versioning the reloading penalty in cycles and GB and GB++ do not need any extra input parametres. There is also a file that the simulator reads to know in how many different and which corespaces it has to perform the simulation. The

output is presented on the screen (if the user choses so), but for large task lists that is impossible to follow. Thus, it is always also stored in a file, as well as other kinds of statistics, like latency per task, downgrading slowdown, how many times and for which tasks the Versioning reloading penalty was applied and total execution time of the whole task list.

4.1.4 Small scripts

Finally there are some small scripts that validate either that the simulator is working correct or simply do something handy for the user of the simulator. Those scripts:

- give the locations of priority tasks in a priority list,
- check if two task lists or two priority lists are identical,
- give statistics on a task list (to see if it's really random or if all tasks appear pretty much with the same frequency in the task list).

4.1.5 Outputs

The outputs and reports have been already described for every component of the simulator, except for the screen output of the Algorithm simulator. The output typically looks like this:

```
Cycle 1: core 4 type chosen for job 1001, duration : 25786
Cycle 1: core 4 type chosen for job 1002, duration : 872
Cycle 873: core 4 type chosen for job 1003, duration : 10623
Cycle 11496: core 2 type chosen for job 1004, duration : 161560
Cycle 11496: core 2 type chosen for job 1005, duration : 14567
...
...
Cycle 941721: core 2 type chosen for job 1013, duration : 3388247
Total sim duration: 4329967
```

Other kind of information might also appear when applicable such as reloading penalty, interrupt, priority, upgrading and generally every decision and action that the algorithm takes.

4.2 Testing methodology

During the development and testing of every algorithm, 3 kinds of tests took place. All three kinds are repeated every time there was a new version of the simulator or a change in an algorithm to validate them.

task	task-ID	GB				Versioning/Basic		
		execution times			pref.	execution times		pref.
		2-issue	4-issue	8-issue		2-issue	4-issue	
adpcm	1001	29750	20229	17025	4	31546	18376	4
bcnt	1002	1003	614	515	4	807	516	4
blit	1003	11202	10337	10334	2	10488	10335	2
compr.	1004	143178	105316	95697	4	128600	98526	4
crc	1005	14054	10056	9209	4	11037	9208	2
des	1006	80272	50562	37729	8	73550	45248	8
engine	1007	798108	626721	583252	2	671589	572852	2
fir	1008	640728	481976	421147	4	552000	448218	2
g3fax	1009	788241	538547	464712	4	562866	469166	2
jpeg	1010	1897159	1339742	1130842	4	1599351	1220330	2
pocsag	1011	34910	22605	17024	8	24845	18018	4
qurt	1012	39644	29880	26053	4	35334	27781	4
v42	1013	3077053	2285802	2006688	2	2310467	2026995	2

Table 4.1: The benchmarks used to create the testing task lists.

4.2.1 Specific microtests

Initially every algorithm is tested with a small task list running each of the 13 tasks of Table 4.1 (which as already mentioned are supplied by [26]) only once. There are several versions of those microtests, each of which is testing a specific situation to see if the algorithm is indeed responding to it as expected. For example receiving a 4-issue task when there is only a 2-issue core free or a task list that should upgrade after one next to it is done running.

4.2.2 Short-run testing

After passing the mircotests algorithms are tested using small random task lists which are checking how the several algorithms compete when there are not so many tasks to execute. Such scenarios simulate the situation of a platform where the MCP does not receive tasks constantly, but in waves. In this phase only total execution time of the task list is recorded and utilisation of AlgA and RAlgA. Four random task lists are always tested and only the averages are presented. All task lists are the same when it comes to the coming order of the tasks but the preferred cores are not the same, since the execution times differ between binaries compiled for Versioning or Basic and Binaries compiled for GB and AlgD. The execution of the same task list evolves differently under each algorithm, resulting is totally different outputs/historics.

4.2.3 Long-run testing

The final tests are the long random lists. Those lists typically consist of 18.000-20.000 tasks. It is the 13 known tasks thrown in the task list randomly but all in about the same rate. Such long tests target to investigate the performance of the algorithms in long usage of the platform with quite a lot of workload (tasks constantly coming for execution). The more the tasks the more cases/circumstances the algorithm could have to deal with in reality are covered.

4.3 Conclusion of Chapter 4

In this chapter we presented the simulator which we had to build to test the developed algorithms, its components, the inputs and outputs of these components as well as their parametres. We are now ready to present the results of the experiments that took place on this simulator.

5 Results

After having defined our algorithms, given enough background to understand both them as well as how ρ-VEX and ERA platforms work and presented the simulator that we developed, we are ready to test the algorithms and analyse the results, find out their weaknesses and their strong points and see how they perform with each other.

5.1 Crosspoints

In order to research the characteristics and the behaviour of the several scheduling algorithms in a systematic way, we detected several parametres and metrics. To investigate the effect of those parametres in the performance of the algorithms we keep for each test all parametres but one fixed/constant and we alter only one to see how that affects the metrics. We expect to find tendencies which change at specific crosspoints for each parametre. Our target is to detect those crosspoints.

The parametres are:

- corespace (number of resources available)
- task list window
- reloading penalty
- GB execution times (for instance 40% and 10%)
- priority density/percentage

Some metrics are:

- total execution time of the whole task list
- speedup or slowdown
- utilisation
- average execution time of a task
- latency (overload execution time)

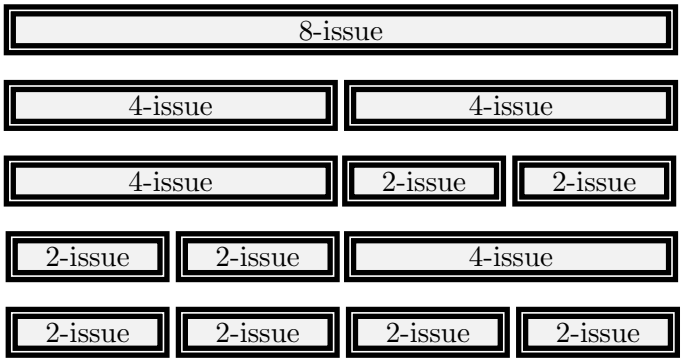


Figure 5.1: The 5 different situations in which a corespace 8 ERA MCP can be found.

5.1.1 Corespace

Corespace is the available resources we have to implement ρ-VEX cores. With the assumption that a 2-issue core takes up 2 units of corespace, a platform with corespace 8, as we saw in Section 2.1 can accommodate either one 8-issue core or two 4-issue cores or one 4-issue and two 2-issue or four 4-issue cores (Figure 5.1).

It is expected that the more space we have to implement cores the faster our system is going to be. Either because it is going to be able to implement more cores and thus run more tasks simultaneously or because it can implement bigger cores and thus run the tasks faster. For corespace 8 to 128, Figure 5.2 presents that speedup of each algorithm to RAlgA, with the exceptions of the All-8 algorithms which present the speedup to their original algorithm (AlgBall8 to AlgB and AlgDall8 to AlgD).

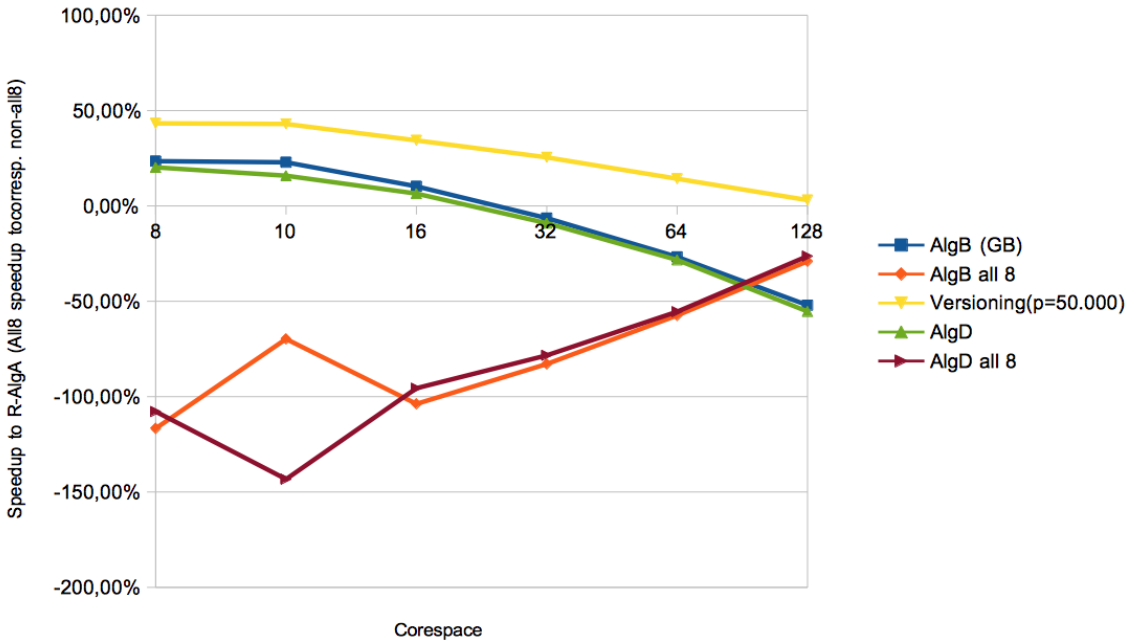


Figure 5.2: The effect of the corespace in the performance of the algorithms.

All algorithms are initially faster than the Basic algorithm (except for the All-8 versions), but as the corespace grows confirming our expectations. The reason GB and AlgD are doing worse than RAlgA is because their binaries are less optimized for cores their running on and thus their execution times are worse than those of RAlgA. As the downgrading instead of bubbling starts happening less and less it stops balancing the worse execution times, which start becoming a burden and eventually drive GB down. But such high scorespace sizes are unrealistic with current technology, especially when we are talking about embedded systems, so in the rest of the research we will not bother with corespaces larger than 16. Note that all tests that we are running from now on are using long task lists of 20.000 or so independent tasks.

One last thing to comment on this graph is the behaviour of All-8 algorithms. Those algorithms are doing worse than all the others because they are trying to run everything on 8-issue cores. And they succeed wherever the corespace is a multiple of 8. But that lead to worse total execution time. Generally upgrading is good for a task, but not good for the other tasks. It leads to better execution time for the task that upgrades to a larger core, but other tasks waiting for resources have to wait and start executing later. That increases the latency of the tasks and the total execution time of the task list, since a task that starts executing later will also finish executing later. Although in real systems a task list has no beginning or end, so measuring total execution time is impossible or meaningless, it is though an indication about the latency. And the latency is perceived as slower execution by the user. When a user opens a jpeg file, if the task doing it spends 2 seconds waiting for resources and 2 seconds processing the image, that is equivalent of spending 4 seconds processing the image if the resources where immediately available. For the user there is no difference. We will come back to the paradox of achieving worse performance by upgrading later on again.

Corespace 10 is the only sample we have between 2 multiples of 8, but there is no doubt the the diamond shape between the line of AlgBall8 and AlgDall8 appears between any corespace sizes that are multiples of 8. With corespace sizes multiples of 8, all the tasks in both algorithms run on 8-issue cores. With other corespaces one task will always not fit. AlgBall8 is outperforming AlgDall8 in those corespace sizes because it downgrades whereas AlgDall8 is leaving a bubble. So RAlgA, AlgD and AlgDall8 are doing nothing useful in those cores whereas all the other algorithms run a downgraded task. The drop in the performance of AlgDall8 is so vast, because it is the only algorithm that definitely will never use that extra 2-issue core, since all tasks are 8-issue and 8-issue tasks in AlgD algorithms are allowed to downgrade only to 4-issue, never smaller.

After this, the further development of AlgD and both All-8 algorithms was abandoned. Alga also appears in the Task list window tests since the Basic Algorithm and AlgD are the only 2 algorithms that use window in the task list.

5.1.2 Task list window

As already explained in Section 3.1.3, a window is the part of the task list visible to the algorithm when it is looking for a fitting task to schedule. That holds only for RAlgA and AlgD (and all their versions). The other algorithms only work with the first task in the task list (FIFO). Since in hardware the resources are unlimited we cannot have a

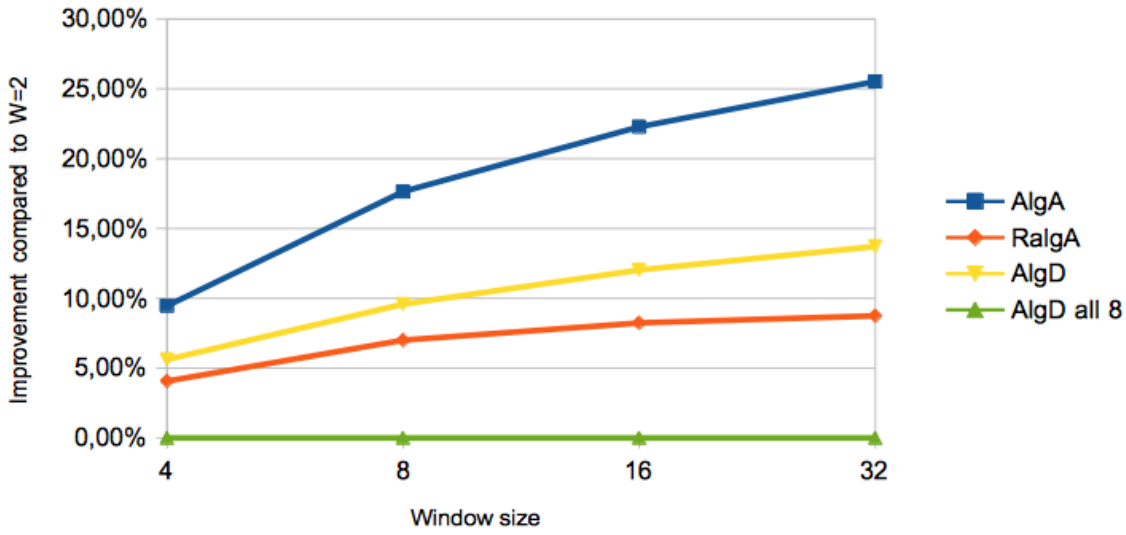


Figure 5.3: Total execution times normalized to W=2.

window of just any size. It is better to look for a balance between speedup and resources. As we see in Figure 5.3, we generally keep benefitting from the increase of the window (theoretically till the window becomes equal to the task list size and RAlgA turns into AlgA), but the benefit is not linear. So a window of 8 will be used in the rest of the experiments.

It is also interesting to see how the window size and the crosspiece affect the utilisation of the resources. Table 5.1 utilisation of AlgA and RAlgA. Note that the utilisation here is the complement of the sum of the bubbles. There are hidden NOPs in the tasks, but we have no information about that. It is obvious from the micro-tests (13-task lists) that the utilisation drops significantly. But this is a pseudo-drop, because while the last task is running, the other cores are empty and the algorithm sees that as a bubble. That ending-bubble has small effect on the utilisation of longer tasks its length is smaller proportionally to the length of the execution historic. The utilisation drops as the corespace increases because more and more cores are empty while the last task is running, so this is also a pseudo-drop. Actually AlgA seems to have an excellent utilisation, and by no surprise, searching the whole task list to find a fitting task, it always finds one. There are practically no bubbles. But such an algorithm is impossible in hardware since hardware resources can support small lists only.

Introducing a Window of size 8, brings a drop in the utilisation, because as predicted it creates bubbles. In micro task lists it has no effect, but that comes not as a surprise, since the size of the Window is almost the same as the size of the task list. What does come as surprise is that it does not have an effect even for RAlgA longer lists. The reason is that probably in those specific test lists, 8-issue tasks, which introduce checkpoints, were relatively often, so that the effect of the Window would not manage to affect the execution. That can by no means mean that the window is useless in RAlgA. In other scenarios 8-issue task might be relatively rare, so the algorithm would need the window so that tasks do not fall in a livelock.

CoreSpace	AlgA	AlgAW8
	13-task list	
8	53,19%	no effect!
10	51,42%	
16	32,61%	
32	16,68%	
	208-task list	
8	98,27%	81,72%
10	99,79%	86,90%
16	96,09%	87,18%
32	84,85%	79,77%
	RAlgA	**RAlgAW:8**
	13-task list	
8	51,45%	no effect!
10	50,63%	
16	32,20%	
32	16,68%	
	208-task list	
8	53,28%	no effect!
10	71,94%	
16	87,56%	
32	79,25%	

Table 5.1: Utilisation of AlgA and RAlgA with and without Window.

There might be a big drop in the utilisation of RAlgA, but from the results we notice that for longer task lists it resists the pseudo-drop that comes with the increase of the corespace. That means that RAlgA performs a more balanced execution, where the bubbles are distributed along the execution and not gathered at the end of the scenario. The poor results of corespace 8 compared to the rest come from the fact that the algorithm really has to wait for all cores to empty, to start running that 8-issue task that created the checkpoint. Whereas with larger corespaces there is always enough space on the side to still do something useful.

5.1.3 Reloading penalty

A characteristic unique to Versioning is the reloading penalty. This penalty comes from the fact that the algorithm calls for replacement of the binary of the task to downgrade, with the binary of the downgraded version. Having an instruction memory that keeps stored 3 versions of every task running or about to run on the MCP is out of question. The new binary that has to come from the main memory. That costs time. Figure 5.4 depicts the performance drop of Versioning as Penalty rises from 5.000 cycles to 400.000 for several corespace sizes. The behavior in all corespace sizes is similar. The penalty really starts having some effect on the performance after the 10.000 cycles and when

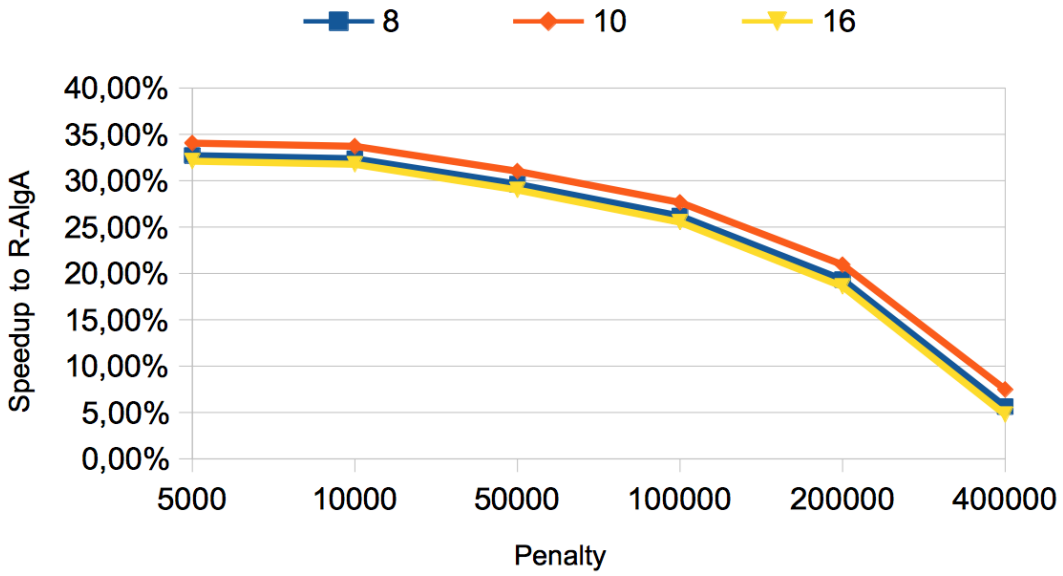


Figure 5.4: Performance drop of Versioning as the penalty increases.

it reaches and exceeds 400.000 it makes the algorithm not worth of the effort and the resources. But is such a high penalty possible?

The penalty is difficult to calculate because it depends mainly on two things: the programme size and the platform. The programme size or the task size (but in terms of number of operations, contrary to the way the term is used in this thesis as the execution time of a task) is known and it is the easy part of the equation. In Table 5.2 the benchmarks and their binary sizes are enlisted. It is normal for the binary size to increase with the size of the core since the perfect parallelisation of the previous binary is impossible, so new NOPs are inserted during the compilation. The biggest binary is the 8-issue version of compress 184.064 bytes, that is 46.016 32-bit OPs. Although the instruction cache is banked, we assume the worst case that 1 op per cycle is coming. That is 46.016 cycles. We proceed to the rest of the experiments with the rough number of 50.000 cycles as the reloading penalty of Versioning. The researcher can improve the simulator by assigning a different penalty for every task according to its binary size or even support different loading rates depending on the platform instead of using the worst case scenario. But that is left as a future research proposal.

5.1.4 GB execution times

During the evolution of this research Brandon & Wong announced an improved compilation for GB, the execution times of which were on average only 10% worse than Versioning/Basic compilation for 4-issue and 20% for 2-issue[26]. But not even the 10% times manage to reach the performance of Versioning in terms of total execution time as shown in Figure 5.5. Looking into the specifics of the penalties will give an idea why. As we saw in Chapter 4, the simulator can extract reports on the penalty. Every time an algorithm downgrades a task there is a performance penalty, since the task will take

task	binary size		
	8-issue	4-issue	2-issue
adpcm	143.136	75.328	50.880
bcnt	108.768	56.304	36.960
blit	115.808	60.368	39.696
compr.	184.064	94.864	60.736
crc	117.280	60.720	39.456
des	124.864	64.832	43.088
engine	146.688	75.536	47.688
fir	117.280	60.656	39.336
g3fax	133.280	68.752	44.648
jpeg	129.088	67.312	44.632
pocsag	142.208	75.376	49.952
qurt	126.592	65.216	41.464
v42	162.784	84.560	54.520

Table 5.2: Binary sizes in bytes for each task and core size.

more time to execute. Versioning has on top of that also a reloading penalty, since it has to bring a different binary to the instruction memory. This one is more crucial than the downgrading overload for Versioning (as seen in Figure 5.6 where the total cycles of penalties paid during the execution of the whole task list is depicted), despite the fact that the simulator recorded that 30% of the tasks were downgraded. GB on the other hand suffered huge downgrading penalty. That comes from the the fact that not only we expect worse performance because of running the task on a smaller core, but

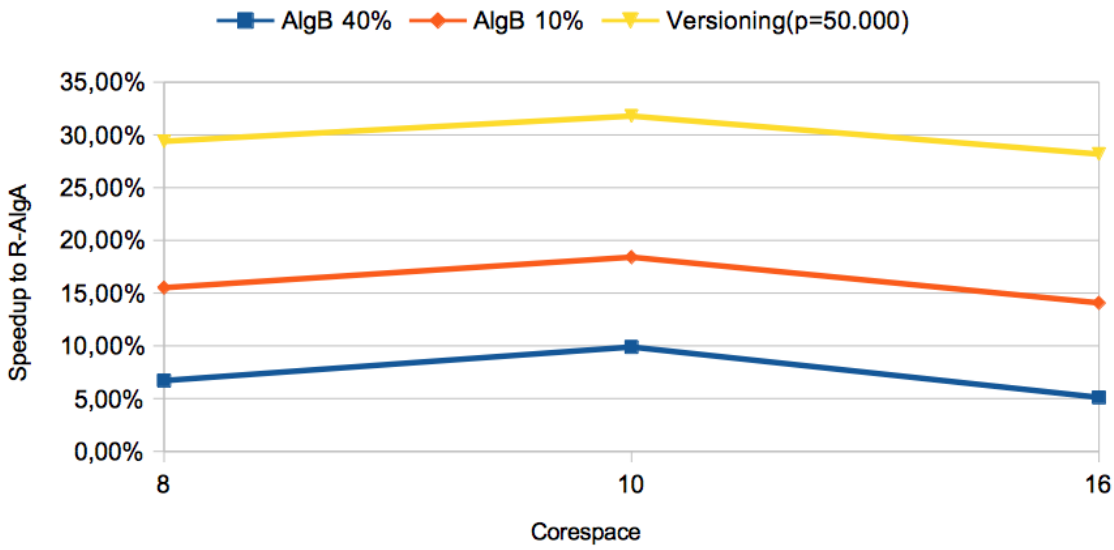


Figure 5.5: Comparison of GB 40% and 10% times and Versioning.

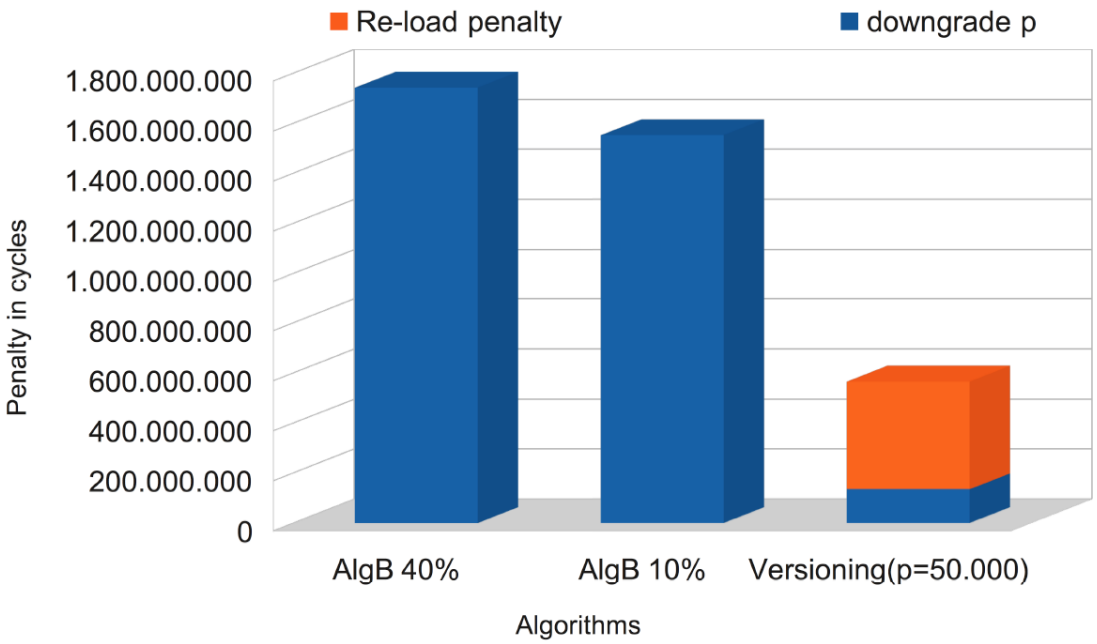


Figure 5.6: Penalty analysis for GB and Versioning.

extra deterioration from the fact that the binary is not optimally compiled for that core. Naturally the question 'Is GB ever going to be better than Versioning?' is created. We are going to try to answer this question later. First we have to take a look at the last parametre: priorities and interrupts.

5.1.5 Priorities and interrupts

At this point we can activate the simulator's priorities and interrupts support. Priorities and interrupts were thoroughly explained in Chapter 3. It only has to be made clear that we are talking about forced single-level priorities. Several priority scenarios were generated and tested, all with the same task list containing 20.000 tasks. The effect of the priority density (the percentage of the tasks that had priority, that is) on the total execution time is depicted in Figure 5.7.

5.1.6 The effect on total execution times

It appears that every algorithm has a slowdown as a result of the priorities. Versioning seems to be very sensitive to priorities. The more they increase the worse the performance of Versioning becomes. The fact that on top of that GB benefits more from the increase of the corespace than Versioning (due to the fact that as seen in Table 3.2 the execution time gain from upgrading is bigger in GB than Versioning), brings the performance of the algorithms closer and closer. For corespace 16 and 20% priority tasks they are almost equal.

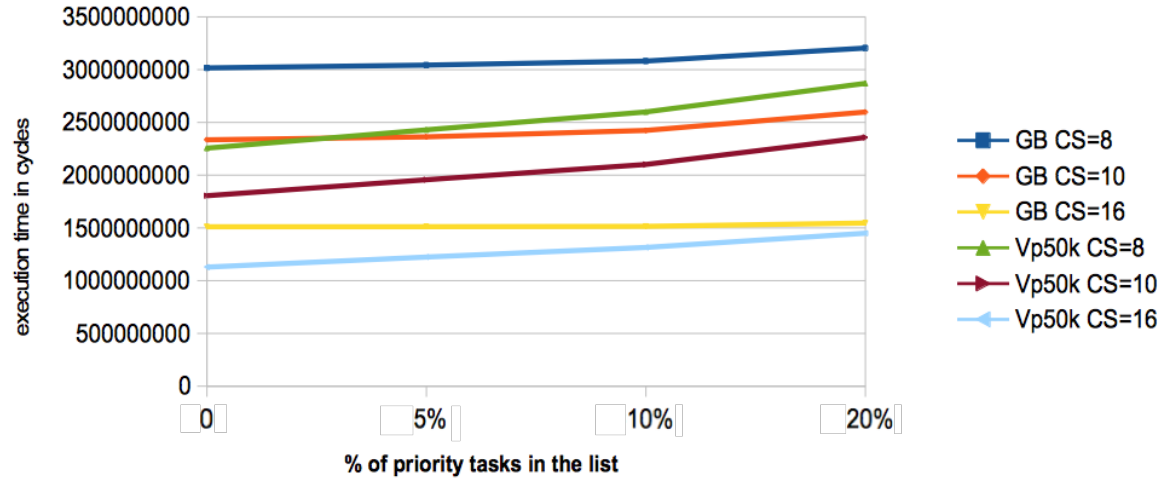


Figure 5.7: The effect of priority density on total execution time of the task list.

It becomes more clear when we normalise the values to no-priority values (Figure 5.8) or when we plot the execution time gain of Versioning to GB (Figure 5.9): GB clearly copes better with forced priorities and interrupts. And the bigger the corespace the faster the difference between the two algorithms disappears. One was already mentioned: GB benefits from upgrading a task more than Versioning. When Versioning gives extra resources to a priority task it wastes resources but it does not enjoy speedup even for the upgraded task (Table 3.2). Another reason is that in GB a task that comes back from being frozen (because of another priority task) might actually have the chance to upgrade when it comes back and finish faster. Versioning on the other hand does not support

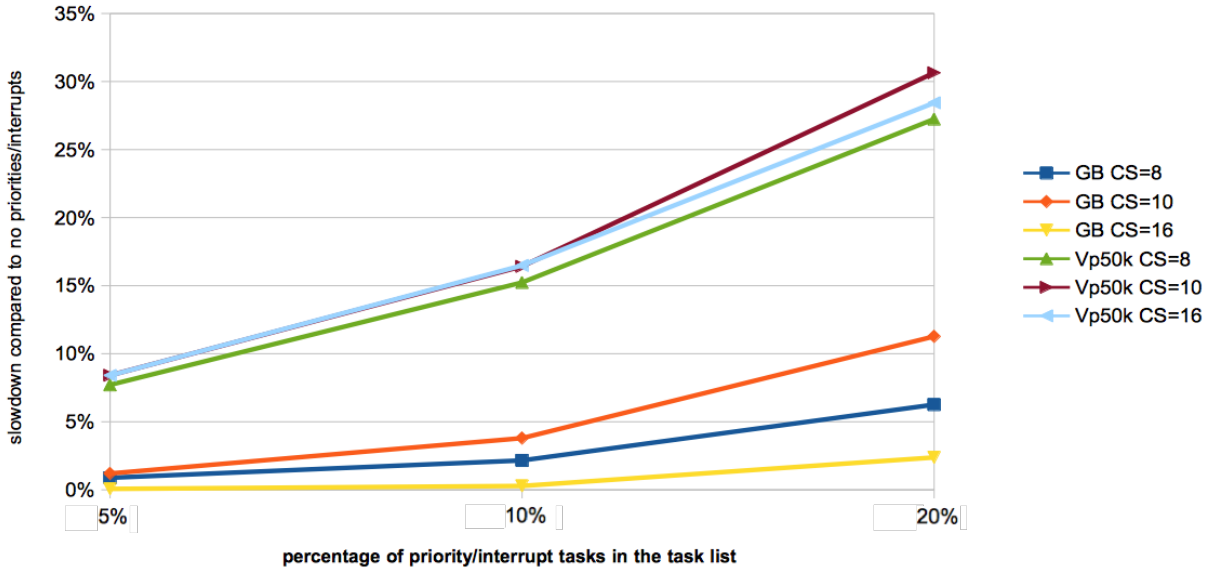


Figure 5.8: Same as Figure 5.7 normalised to no priority execution times.

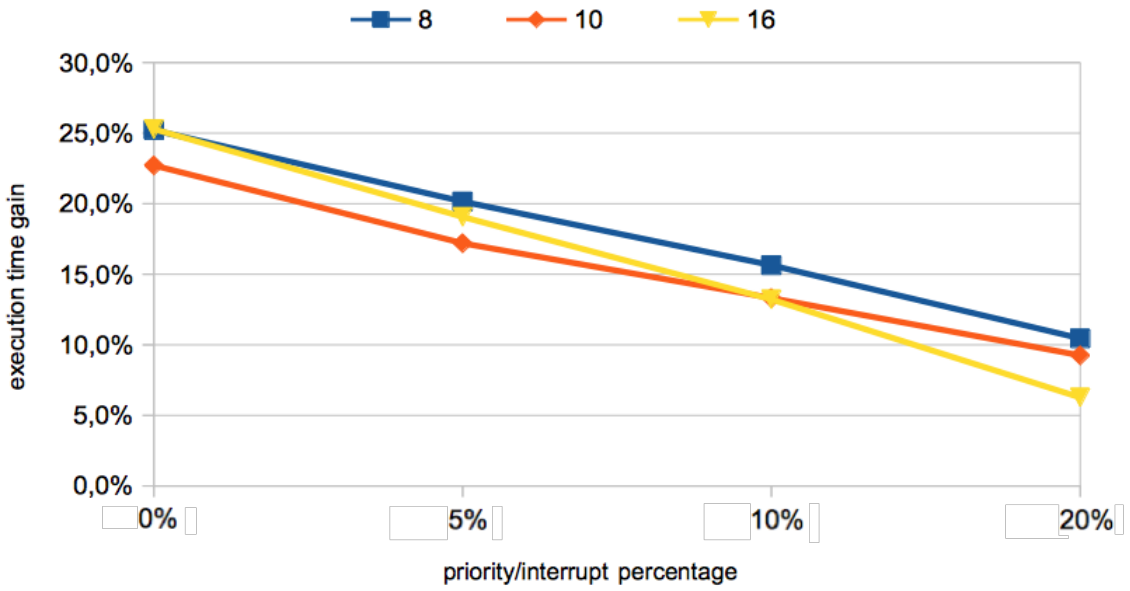


Figure 5.9: Versioning losing its speedup to GB as priority density increases.

upgrading, it would have to start running the task all the way from the beginning.

Finally, GB performs better than Versioning because it can handle an interrupt by downgrading a 4-issue or 8-issue task instead of freezing it. This way the overhead is only the downgrading penalty of the old task, not even all of it, only of the remaining part, whereas in Versioning the overhead of handling an interrupt is the whole execution time of the interrupt.

5.2 Task latency

Achieving shorter execution time of the whole task list is an indication of smaller latency for each task, but no guarantee. The simulator was, thus, improved to report on task latency. It keeps an eye on every task from the first moment they come in contact with the Task scheduler, until they have declared they are done with execution. That might include waiting for execution or being frozen or being downgraded or upgraded. Task latency is everything that does not include execution time.

In this part of the study lots of results were produced, quite a large amount of graphs, since we are taking a look on each task independently. Luckily the tasks form clusters in which they behave similarly, so not all the results have to be presented, but only the results of one task of each group/cluster. Seeing some characteristics of the tasks will help the reader understand the results better.

5.2.1 The nature of the tasks

The execution time of the tasks have already been presented in the tables of Chapter 3, but a visual representation gives a better understanding of the huge differences between

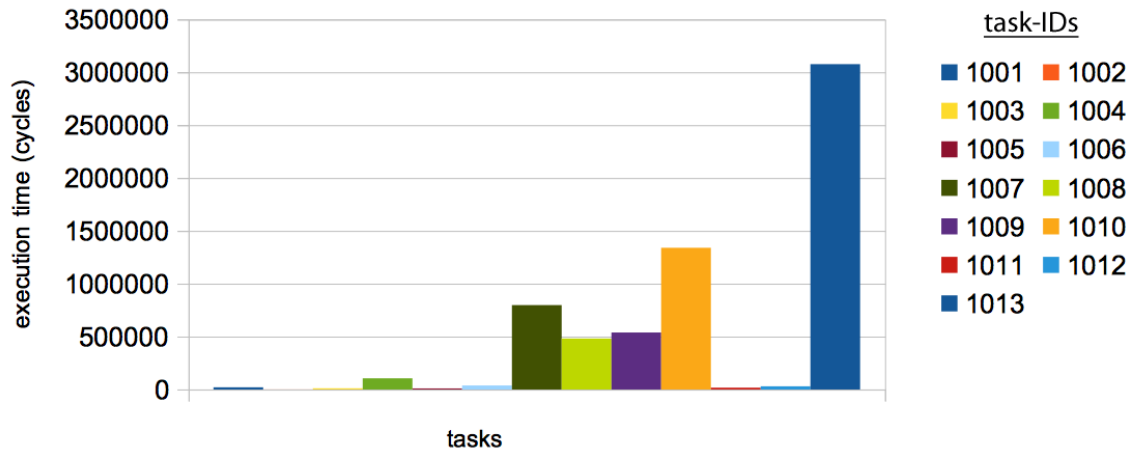


Figure 5.10: The GB preferred execution times.

them. In Figure 5.10 the preferred/optimal execution times of the tasks for GB are presented and in Figure 5.11 those of Versioning/Basic.

For GB we easily spot 2 or 3 groups of tasks. Tasks 1 to 6 and tasks 11 and 12 are small tasks. We can either group the rest (7, 8, 9, 10 and 13) up and call them the large tasks or consider tasks 7, 8, 9 and 10 as medium task and distinguish 13 from them as the only really large task.

For Versioning and basic the situation is pretty much the same. Perhaps Task 10 can be considered a large task and not a medium one, forming a separate group with task 13.

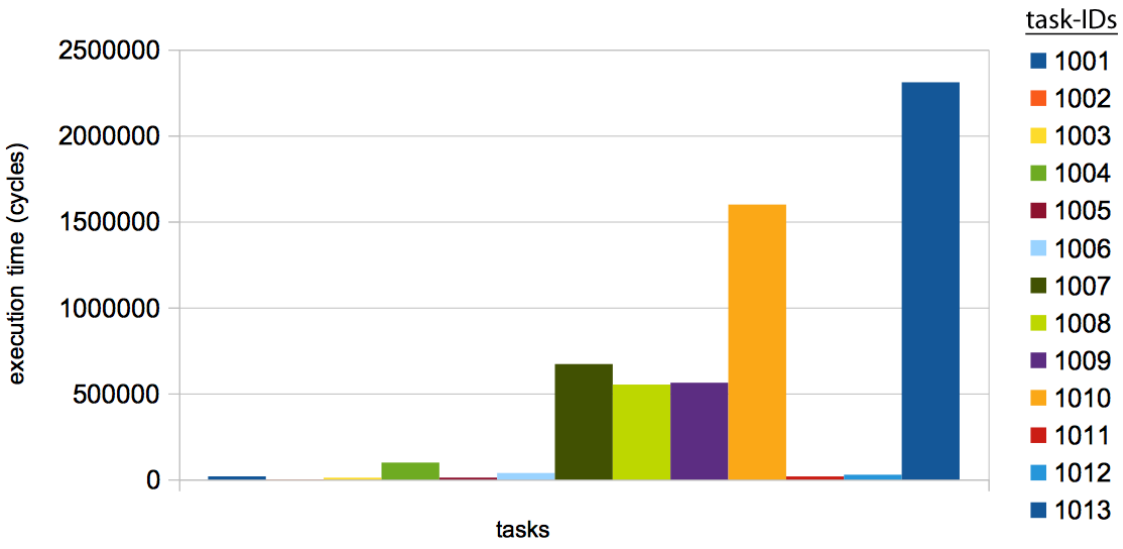


Figure 5.11: The Versioning/Basic preferred execution times.

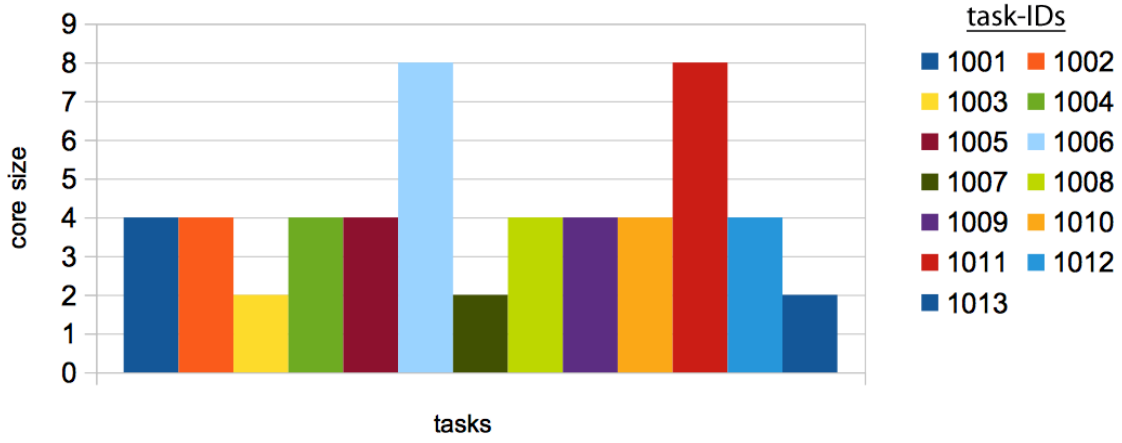


Figure 5.12: Preferred core sizes per task in GB.

Another characteristic of the tasks is the preferred core. The preferred core is chosen by starting from the 2-issue times and upgrade to a higher core only if the speedup is more than 25%. The percentage is just chosen as the middle between no gain and the theoretical maximum which is 50% (see Section 5.4.1), but it can be a subject of research.

In this way we see that for GB tasks 3, 7 and 13 are 2-issue, tasks 1, 2, 4, 5, 8, 9, 10 and 12 are 4-issue and tasks 6 and 11 are 8-issue. Versioning on the other hand has only one 8-issue task. That is task 6. Further tasks 1, 2, 4, 11 and 12 are 4-issue and tasks 3, 5, 7, 8, 9, 10 and 13 are 2-issue. Task 6 actually does not qualify for being an 8-issue task, since its improvement is only 17% (Table 3.2), but considering Versioning

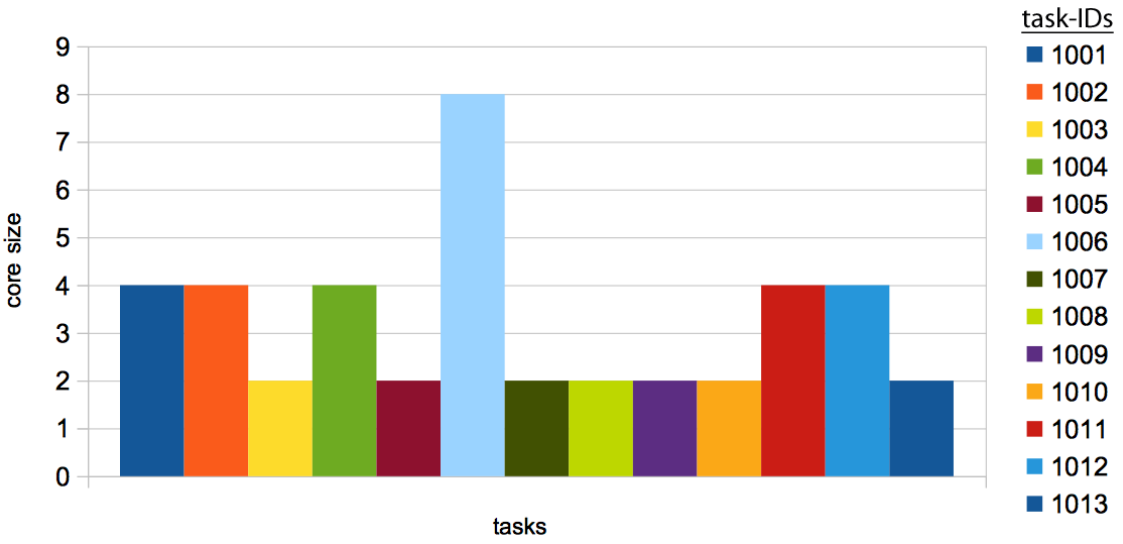


Figure 5.13: Preferred core sizes per task in Versioning/Basic.

and Basic would have stayed without any 8-issue tasks and since its speedup stand out compared to the rest of the tasks, we have chosen to consider it an 8-issue task.

We already see that Versioning and Basic have a lot of 2-issue tasks. That could explain why we found such small downgrading and reload penalties in the penalty analysis (Figure 5.6). Even task 6, an 8-issue task, being so small (in terms of execution time) it does not affect so much the development of the execution.

5.2.2 Latency and priorities/interrupts

It is high time we checked the results from task latency. Figure 5.14 presents the average execution time per task which includes both the net useful execution time as well as the task latency for GB and specifically corespace 8. The values are normalized to the no-priority values. In other words we see the slowdown of every task as the priorities increase. Since task 2 has a huge difference, it dominates the graph. It is no surprise; task 2 is the smallest task, only a few hundred cycles long. We are going to take it out, so that we can zoom in the area where most of the tasks are (Figure 5.15).

We see clearly that small tasks are more sensitive to interrupt/priority task density, since all the small tasks are exploding in the graph, whereas bigger tasks are gathered at the bottom. Core size does not seem to play a significant role, since our 8-issue tasks did not manage to benefit from the priorities. The fact that they are small tasks seemed to play a more important role. Unfortunately we do not have any large or even medium tasks with 8 as its preferred core. But task 4, which is indeed the biggest among the small tasks, but still 6 to 20 times smaller than the large tasks still manages to resist priority density, and that might be because it benefits from the upgrades, being a 4-issue task. The situation is pretty much the same for corespace 16 (Figure 5.16). The only difference is that the magnitude (y-axis) is smaller.

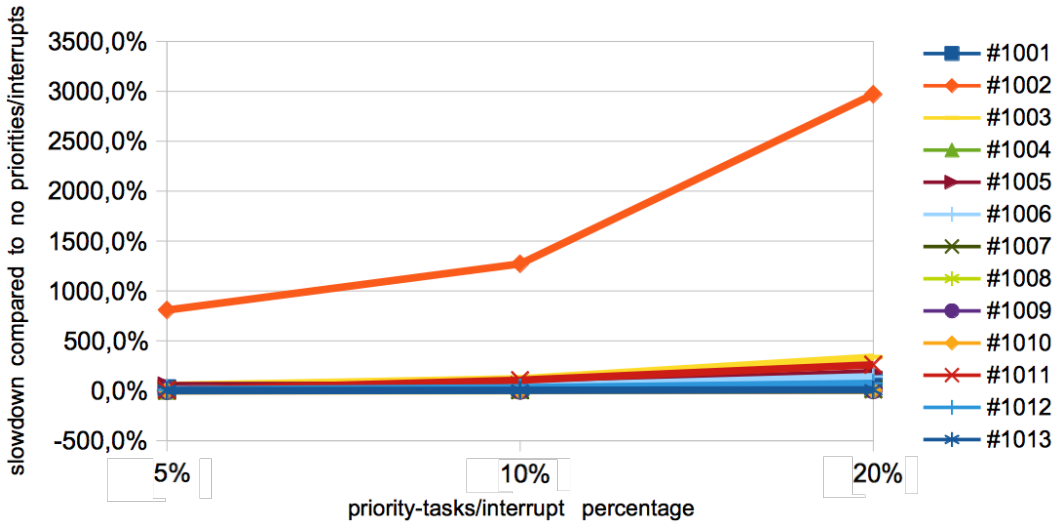


Figure 5.14: GB execution time + latency slowdown to 0%-priorities for corespace 8.

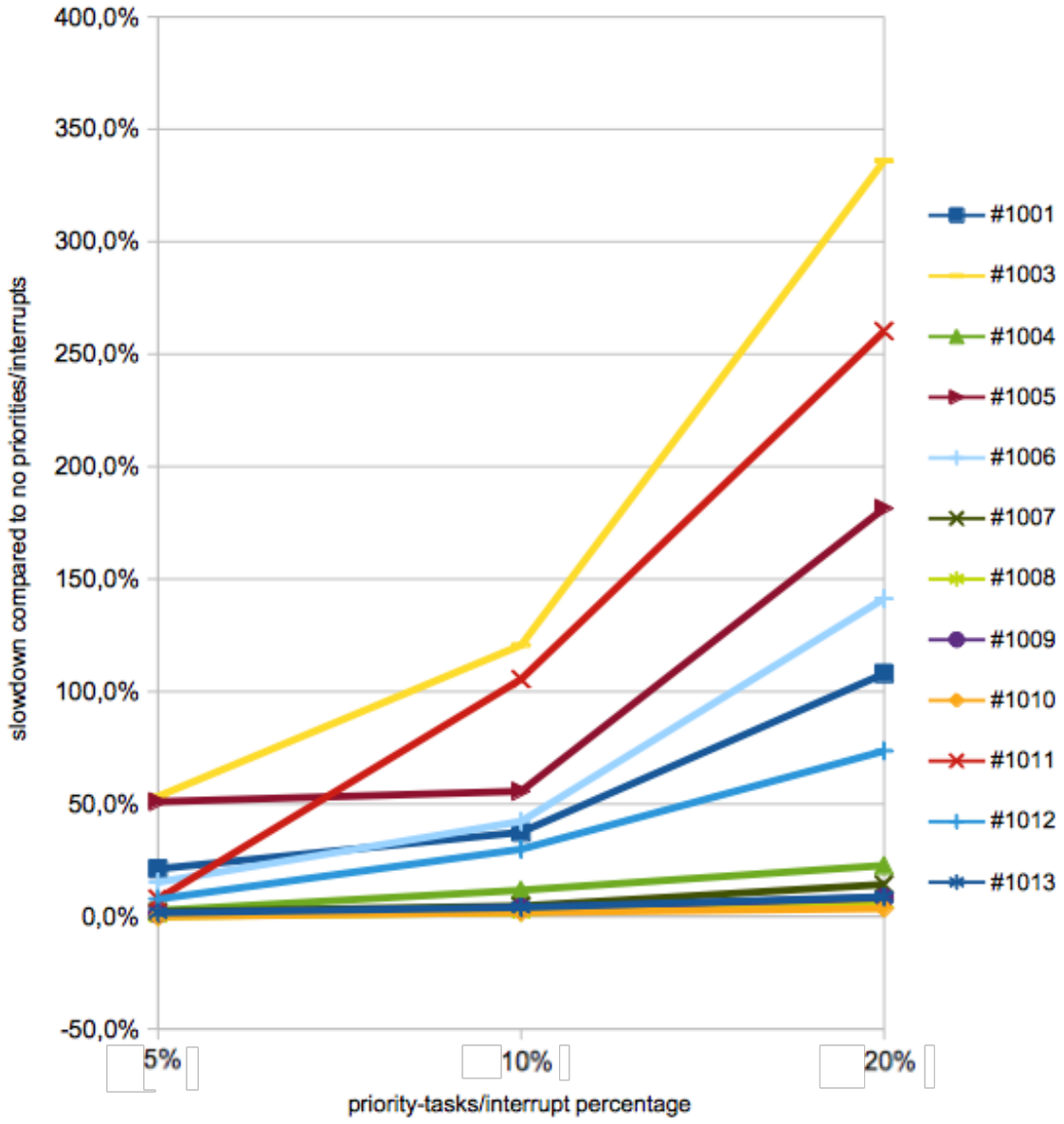


Figure 5.15: Same as Figure 5.14 without task 2.

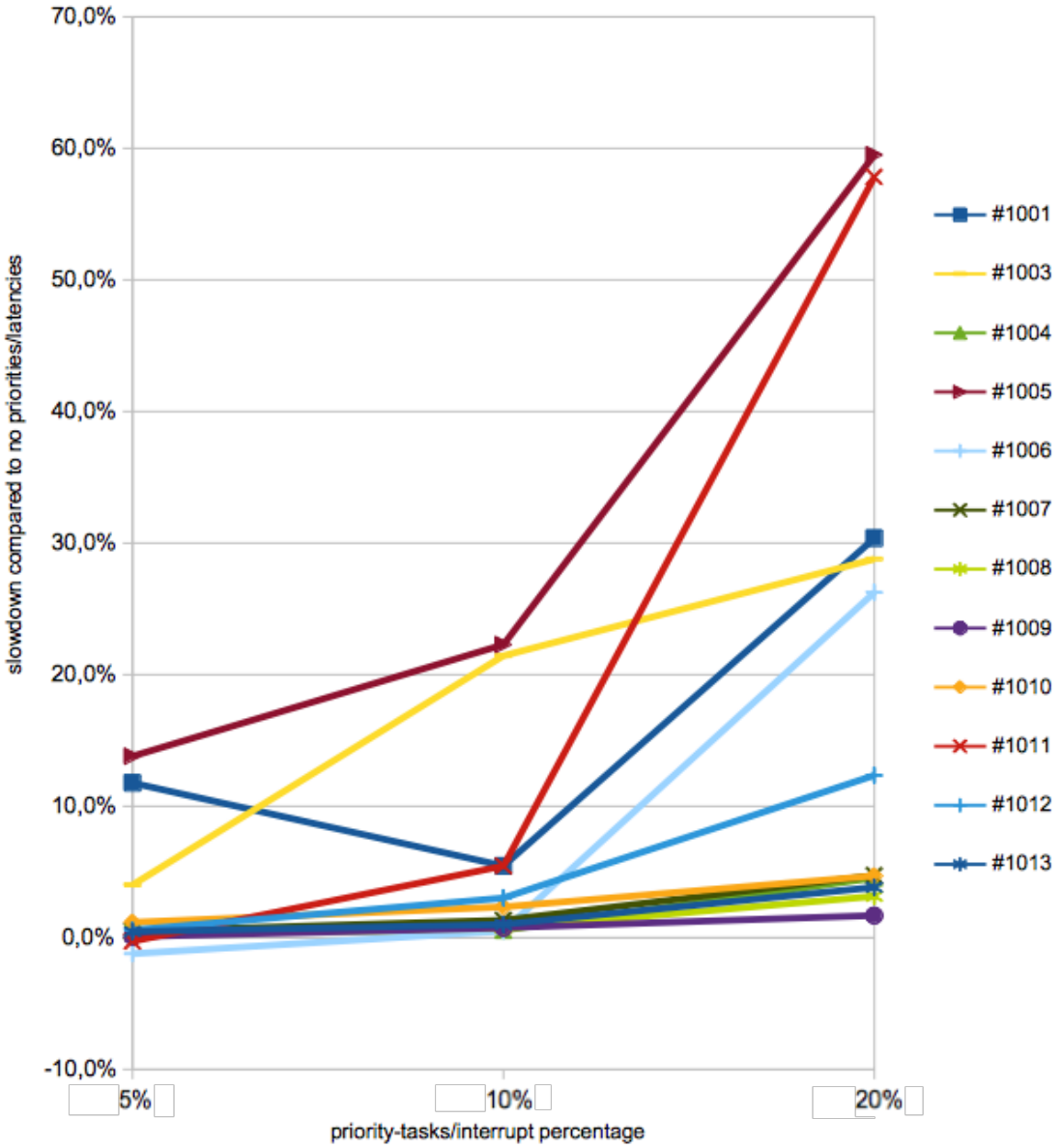


Figure 5.16: Same as Figure 5.15 for corespace 16.

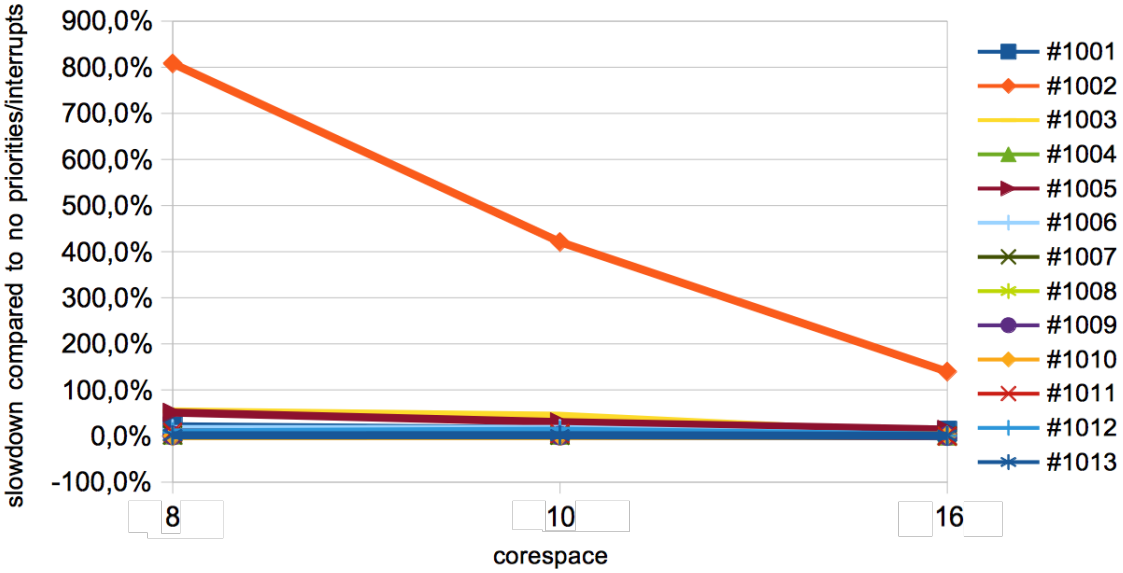


Figure 5.17: GB execution time + latency slowdown to 0%-priorities for priority density 5%.

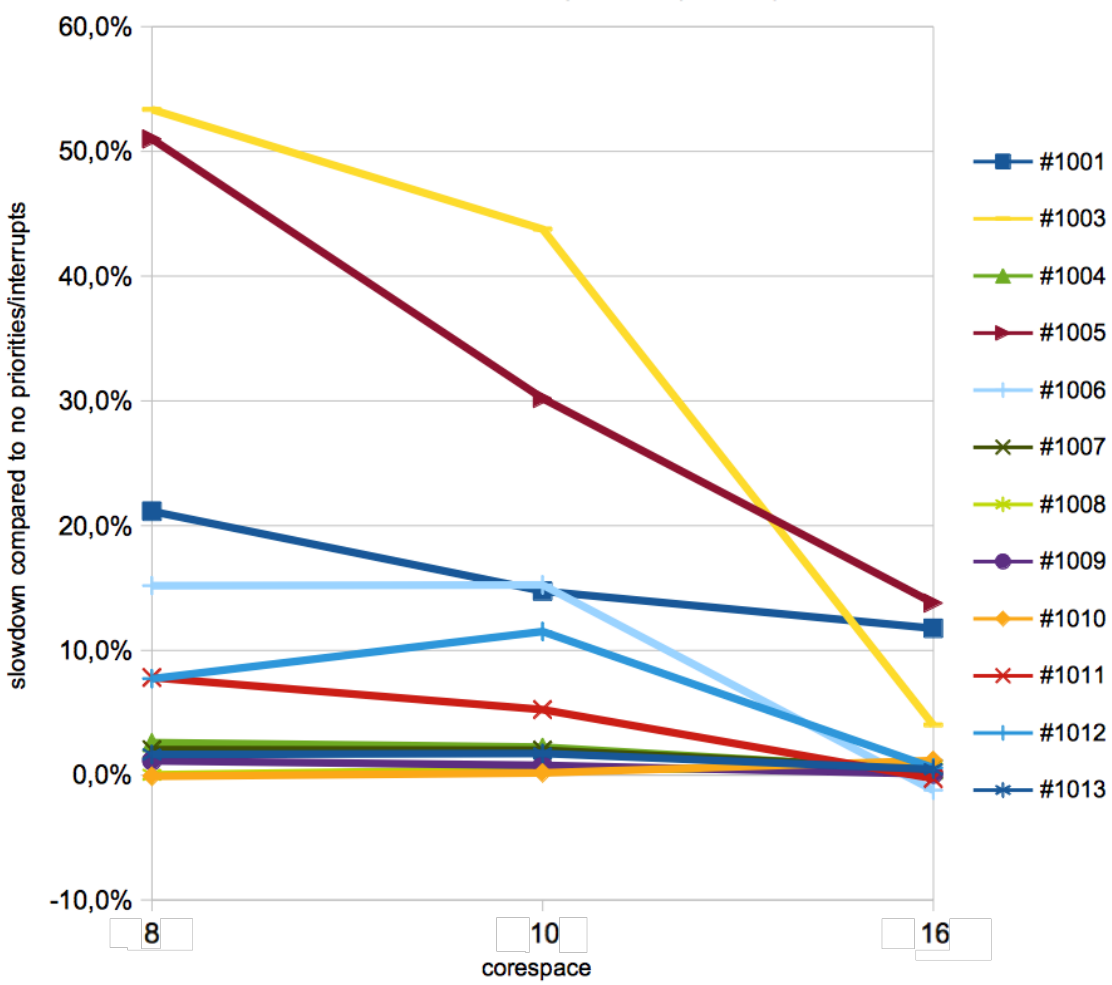


Figure 5.18: Same as Figure 5.17 without task 2.

Now we keep priority density steady to see the reaction of every task to the increase of the corespace (Figure 5.17). Once again we will have to remove the tiny task 2 to see the details at the bottom of the graph (Figure 5.18). Corespace really helps GB cope with priorities. We see the slowdown effect of the priorities even on small tasks dropping a lot at corespace 16. That happens because, since the tasks are being executed in a much wider area, the chances that a priority affects a small task is smaller. In an MCP of corespace 16 (8 2-issue clusters that is) a priority that expands from cluster 7 to cluster 8 will only affect the tasks that will make use of cluster 8. It even benefits tasks that make use cluster 7 (since it is going to release it sooner than if it did not upgrade) and it does not affect any task that will run in clusters 1 to 6 at all. The big slope we see on figures Figure 5.18 and 5.19 is actually the magnitude drop that we notice when we go from Figure 5.15 to 5.16 and vice versa.

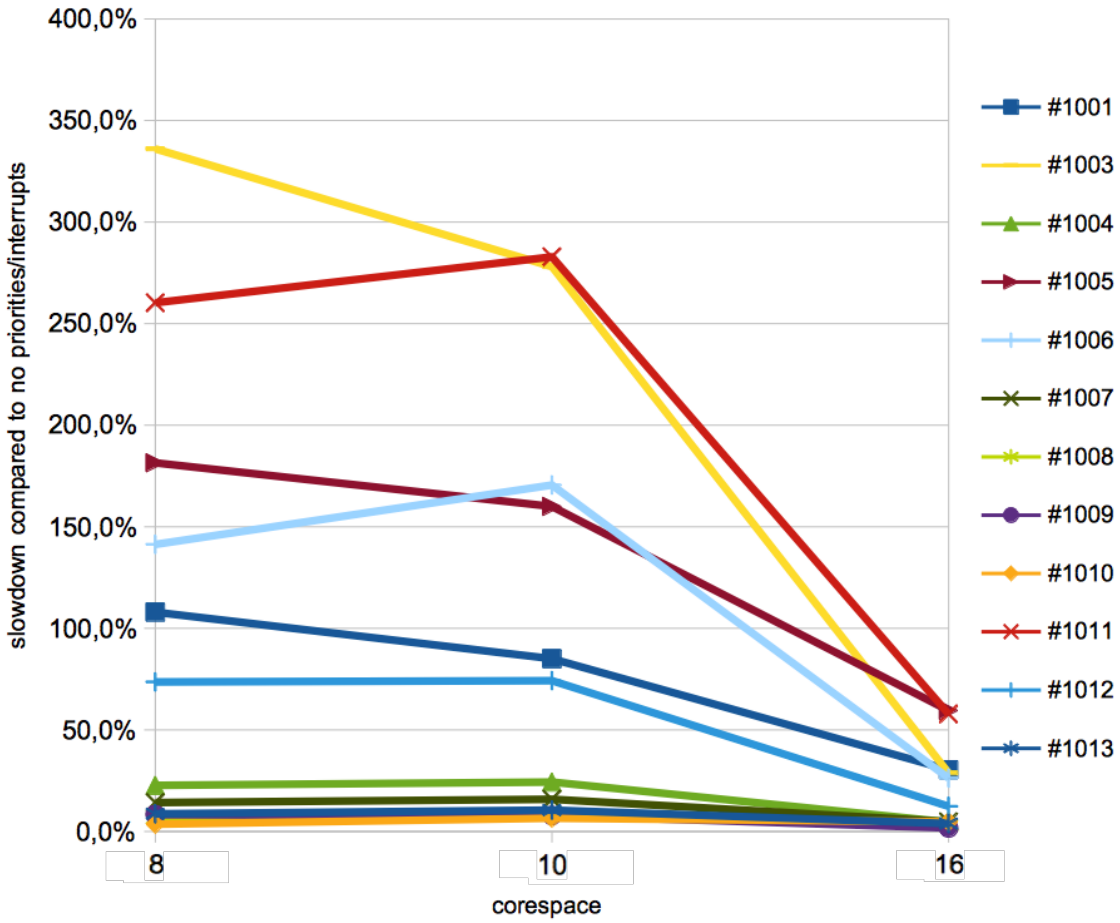


Figure 5.19: Same as Figure 5.18 for priority percentage 20%.

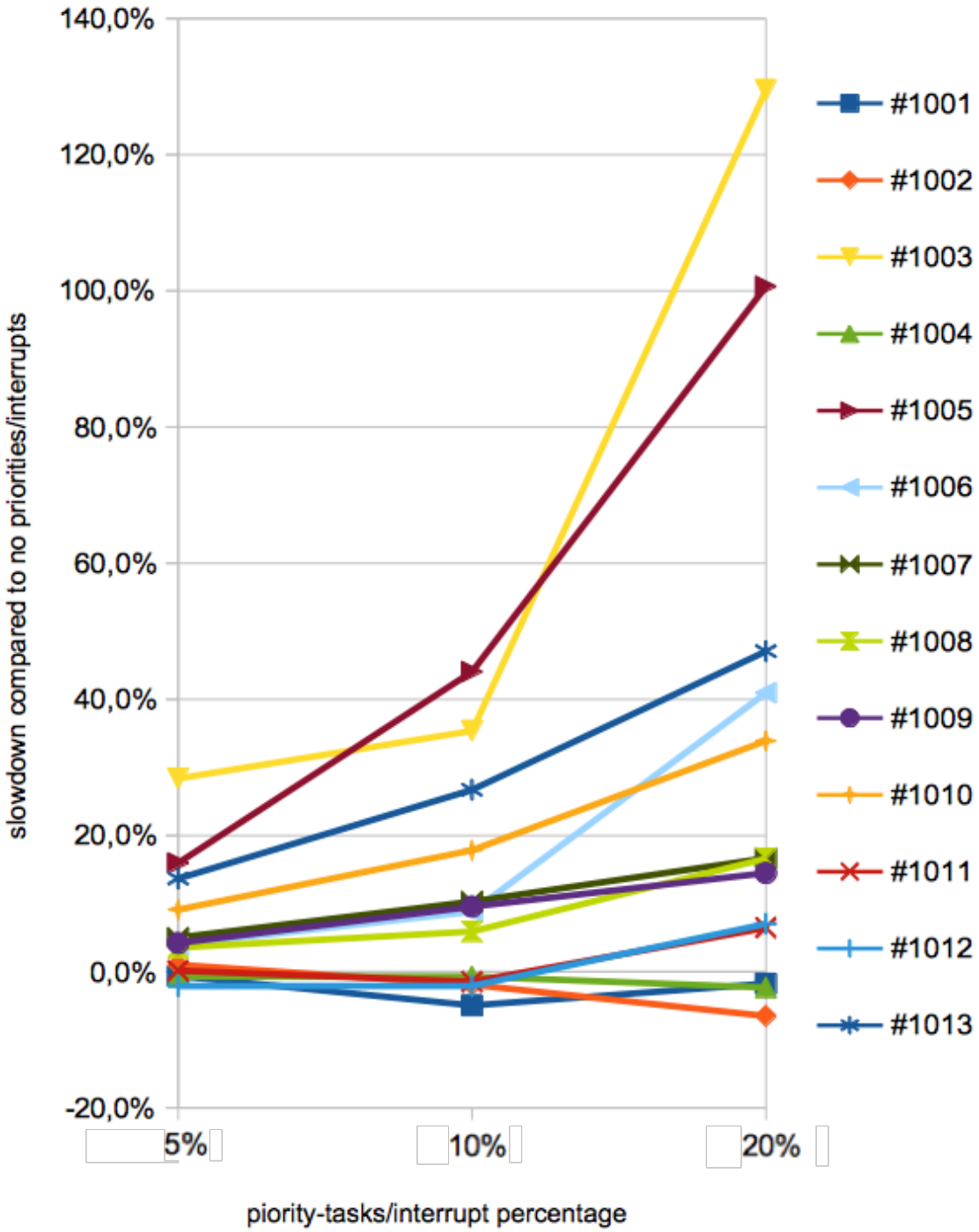


Figure 5.20: Versioning execution time + task latency slowdown to 0%-priorities for corespace 8.

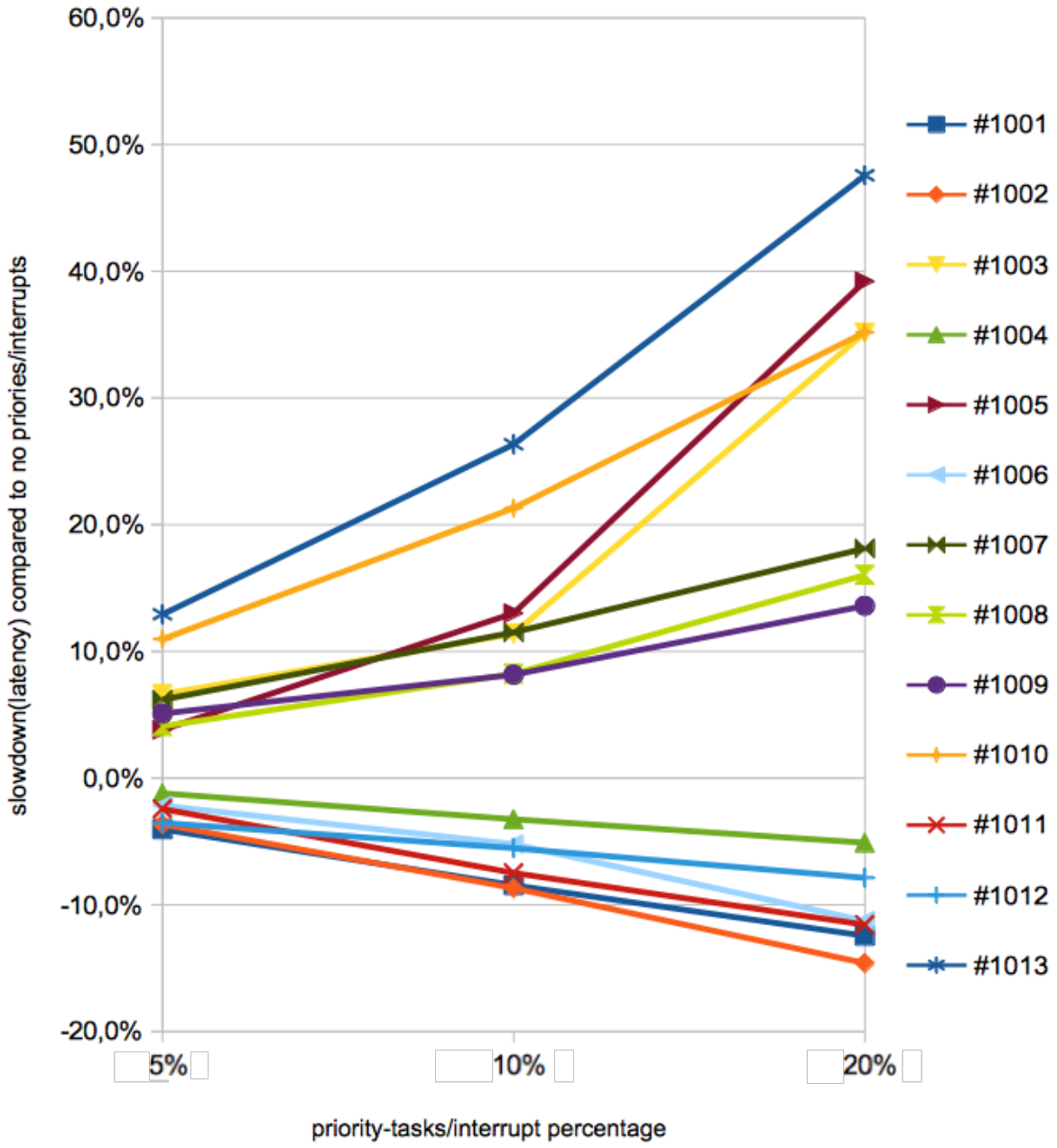


Figure 5.21: Same as Figure 5.20 for corespace 16.

The picture in Versioning is a little bit different. Here it is not about big and small tasks. It is about core sizes. Figure 5.20 and 5.21 it is obvious that 2-issue tasks suffer the most from latency. 4-issue tasks benefit when they are priority tasks, because they will run faster. 2-issue tasks cannot do that, their only benefit from being priority tasks is that they will not have any latency. Especially in Figure 5.21 the speedup that some tasks are achieving is obvious. That speedup though is only among small tasks, so it does not have a big effect on the total execution time of the task list. In contrast, task 13, which is the biggest task, suffers a lot more latency in Versioning than in GB. One reason could be that although it is the biggest and most dominant task, like its GB version, it is still 25% smaller than it. Which means that it gets more affected by the priority of other tasks. On top of that, the other "big" tasks of Versioning are larger (take longer to execute) than in GB also increasing their priority effect on other tasks. A third reason might be that GB is better than Versioning when it comes to interrupts. GB can downgrade a 4-issue task to make space for the handling of an interrupt. This way all tasks keep running. Versioning cannot do that. It will freeze a 2-issue task to handle the interrupt and task 13 is a 2-issue task. Looking at Figure 5.21 and considering that task 13 is the slowest and most dominant task in the task list, we can conclude that it is the main reason why Versioning performance drops down to GB levels at Figure 5.7 or 5.9, despite the speedup of the 4-issue tasks.

One wonders why 4-issue tasks manage speedup in Versioning and not in GB. They probably do, but in GB tasks have another kind of negative effect from other tasks that does not exist in Versioning: upgrading. Upgrading extends an adjacent task when a 2-issue or 4-issue priority task stops running, which does not happen in Versioning. That means the frozen task has to wait also for the upgraded task to finish. Considering most 4-issue tasks are small this causes big latency. We expect more pressure from priority tasks to other tasks in GB also because there are more 4-issue and 8-issue tasks (than in Versioning) which affect the performance of other tasks more than 2-issue priority tasks since they take more space (the effect on adcpm described in Figure 5.36 and found to be fading with the increase of corespace). We can already notice that extra pressure from priority tasks to non-priority tasks from the difference in magnitude of the y-axis of Figure 5.15 and 5.20.

Just like with GB, we are exploring the effect of corespace in Versioning by keeping priority density stable at 5% the one time and 20% the other time. From Figure 5.22 and 5.23 we see that Versioning does not really take advantage of corespace like GB does (Figure 5.18 and 5.19). Only Tasks 3, 5 and 6 seem to benefit for corespace 16. Tasks 3 and 5, being 2-issue tasks have less chance to be affected by another tasks priority in a bigger corespace and on top of that more chance that another task will finish sooner and they can continue their execution in that cluster, rather than the old one. Task 6 is an 8-issue task. In corespaces smaller than 16, the chance is big that even if it is a priority task, it will fall on another priority task that is already running and it will not manage to run. That is something 8-issue tasks suffer from, both in GB and Versioning. For corespace 16, though, 8-issue tasks have bigger chance to find enough (priority-task-free) space to take advantage of their priority.

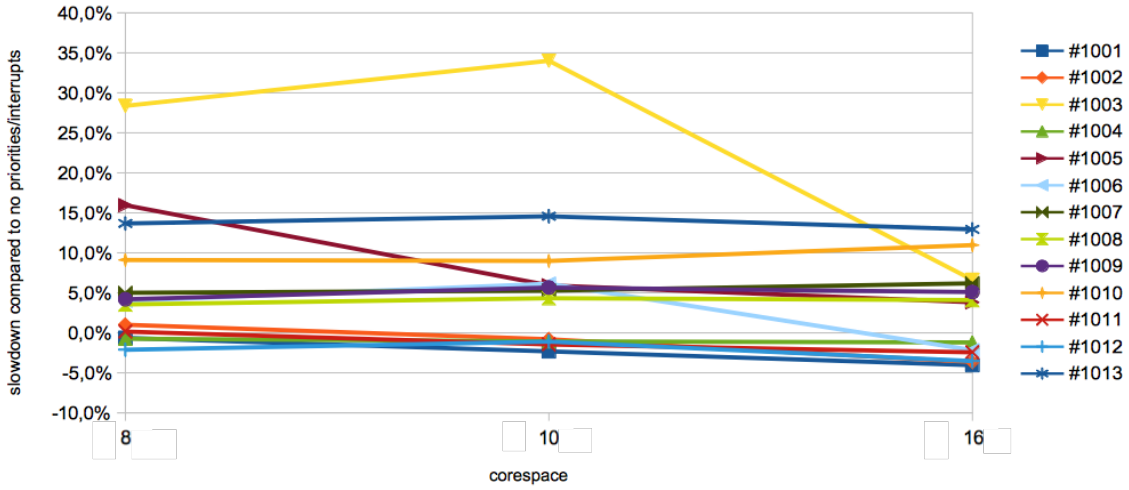


Figure 5.22: Versioning execution time + latency slowdown to 0%-priorities for priority density 5%.

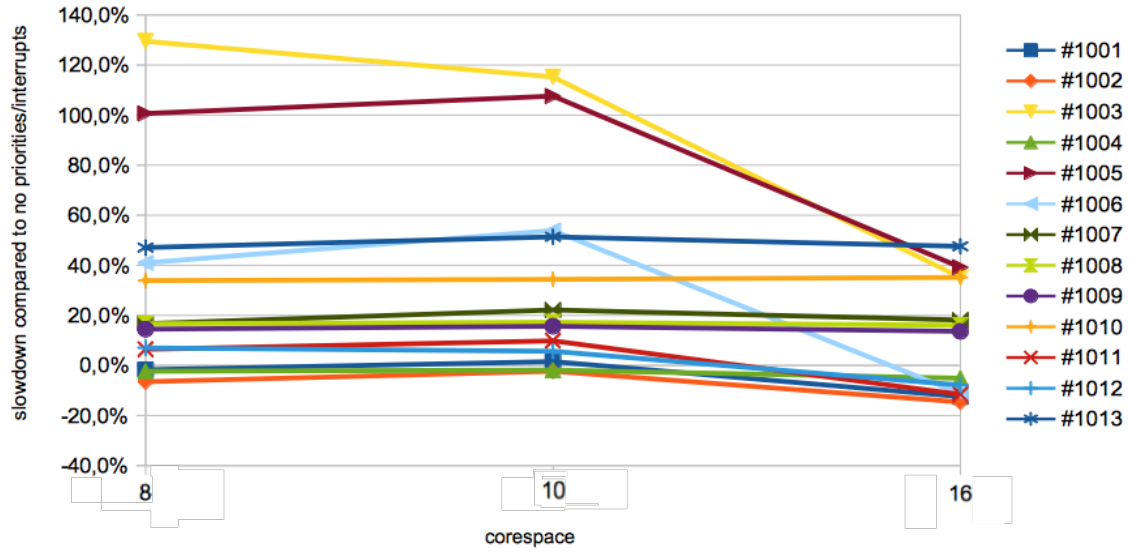


Figure 5.23: Same as Figure 5.22 for priority density 20%.

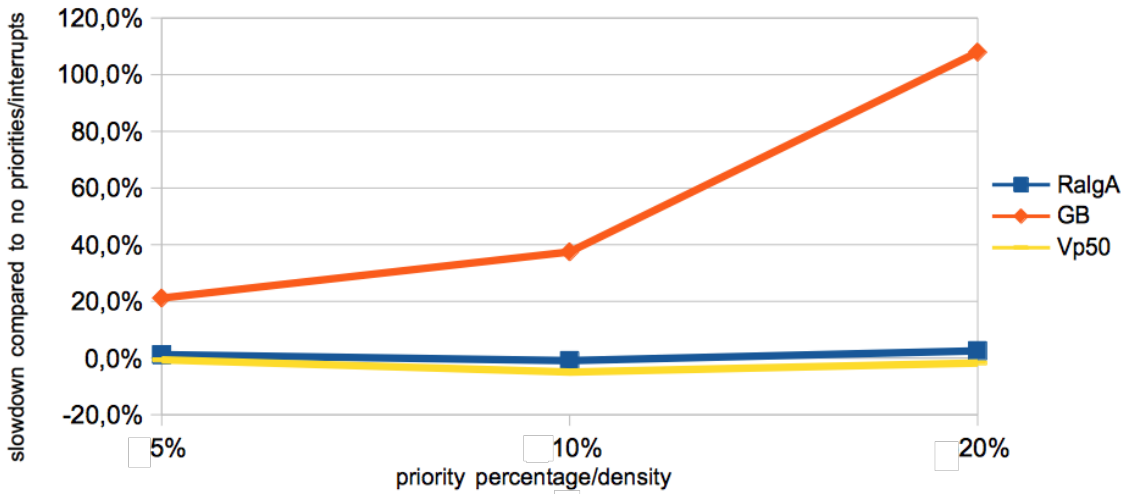


Figure 5.24: Execution time + latency slowdown of task 1 for all algorithms compared to no priority times (corespace 8).

The above results become more clear if we focus on two specific tasks. Firstly, taking a more careful look to a small 4-issue task, such as Task 1, we see that it indeed suffers more latency with GB than the other algorithms (Figure 5.24). In corespace 16 (Figure 5.25), however, we see the magnitude of the graph dropping impressively. It confirms that GB takes advantage of corespace, since there is less chance that the task will be affected by other priority tasks, more chance that it will find non-priority tasks to freeze and take over their resources and more chance to find space to upgrade (the last is not true for Versioning). Being a small task, it benefits from the fact that there are not so many 4-issue and 8-issue tasks and it manages speedup for Versioning.

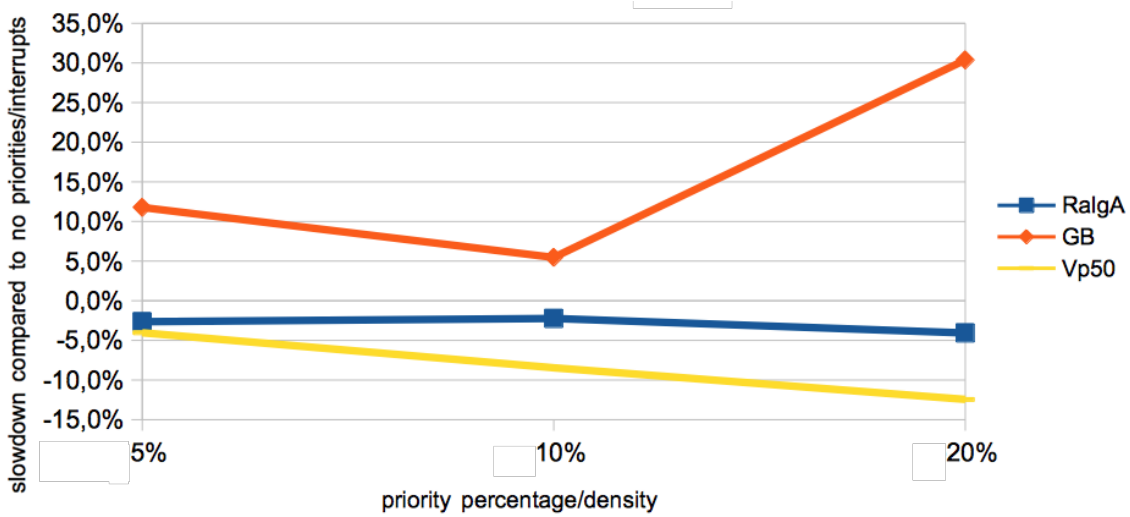


Figure 5.25: Same as Figure 5.24 for corespace 16.

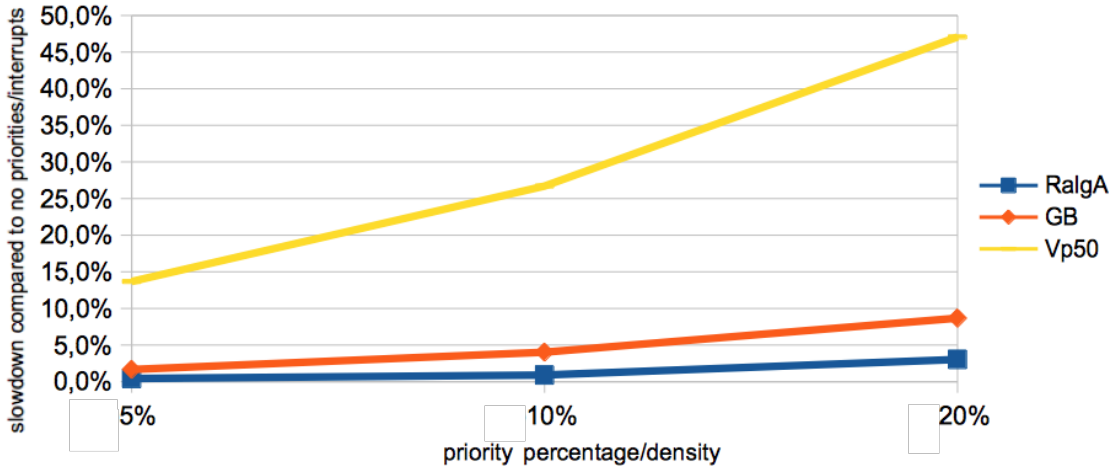


Figure 5.26: Execution time + latency slowdown of task 13 for all algorithms compared to no priority times (corespace 8).

In big tasks, like tax 13, the image is reversed (Figure 5.26). GB is resisting the increase of the interrupts whereas Versioning gives in to them. Once again GB benefits from the increase of the corespace, dropping below 5% for corespace 16 (Figure 5.27), but Versioning not. Those graphs can be misleading, because the algorithms are being compared to themselves. Looking at the raw numbers we notice a reversed image.

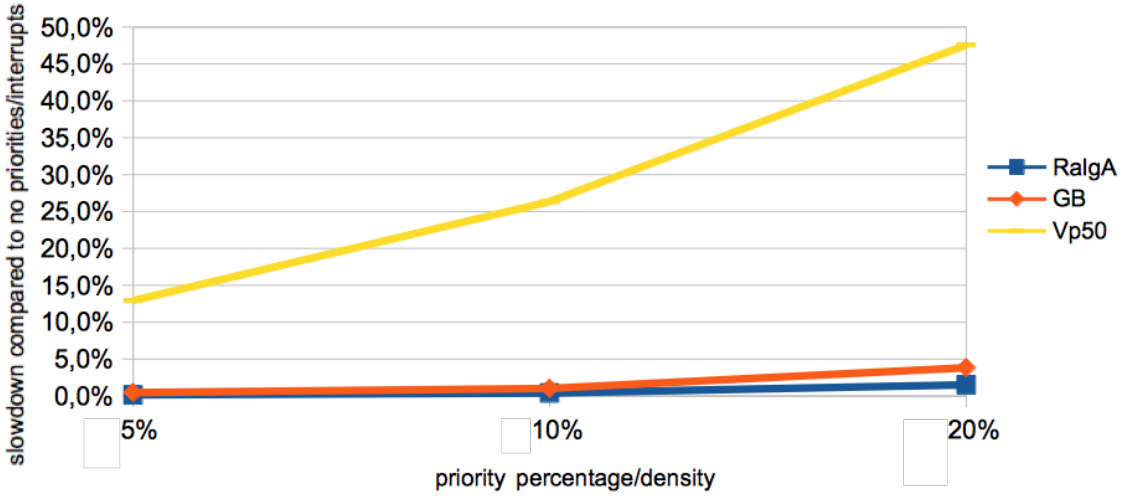


Figure 5.27: Same as Figure 5.26 for corespace 16.

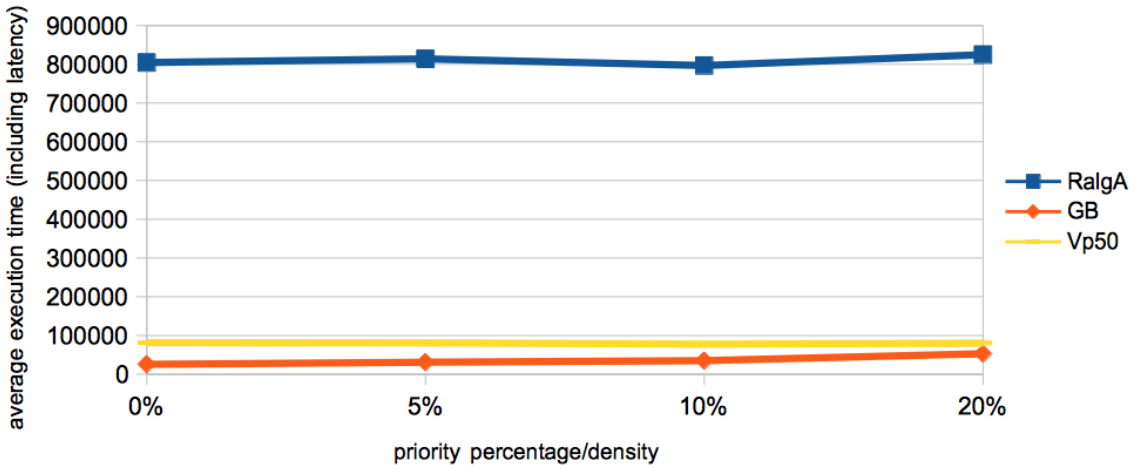


Figure 5.28: Task 1 average execution time (incl. latency) for each algorithm for corespace 8.

Although Versioning seemed to be the winner in small tasks, because of its small slowdown and even speedup (with the RAlgA showing similar performance), the reality is actually the opposite (Figure 5.28). Versioning might respond better to interrupts in small tasks, but its latency is actually worse than GB. RalgA as expected has the worst latencies, since the algorithms just stalls the tasks until there is enough space to run them in their preferred core size. It is more clear for corespace 16 (Figure 5.29). The speedup of Versioning and the slowdown of GB (Figure 5.25) are not enough to make the two algorithms meet. GB is still faster.

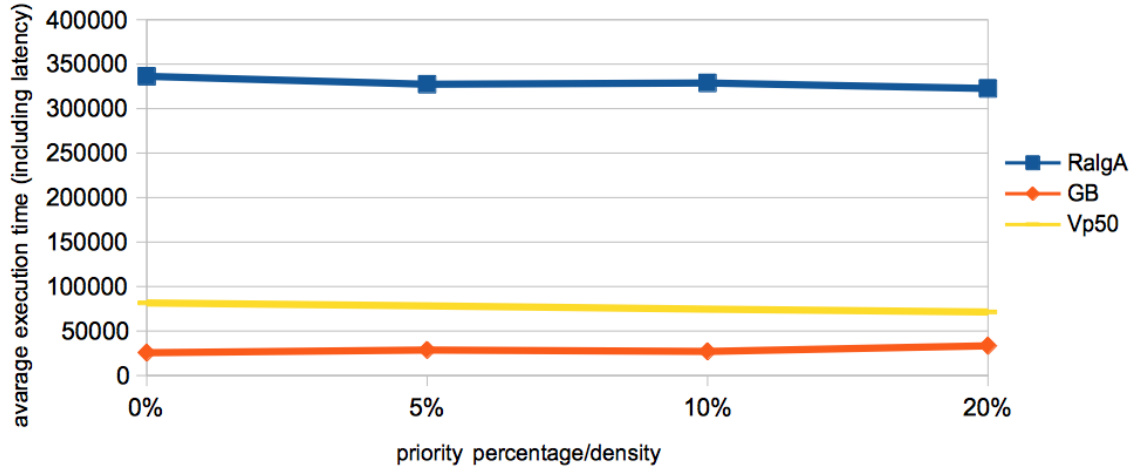


Figure 5.29: Task 1 average execution time (incl. latency) for each algorithm for corespace 16.

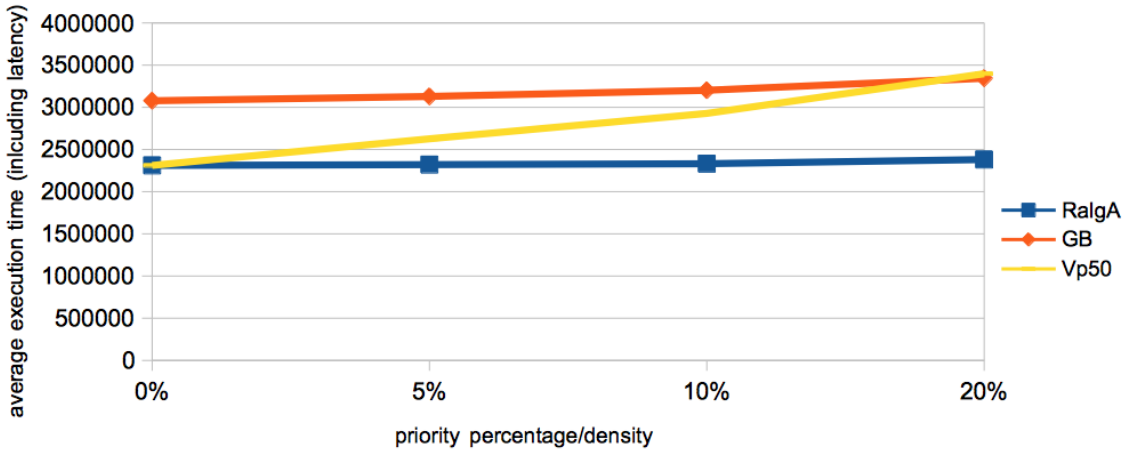


Figure 5.30: Task 13 average execution time (incl. latency) for each algorithm for corespace 8.

The image is reversed in big tasks, such as task 13 (Figure 5.30) too. Although GB is coping better with priorities/interrupts, Versioning starts from much lower levels (we have already mentioned that the Versioning binary of task 13 is 25% faster than that of GB) and it only becomes worse than GB at priority density levels of 18% or higher. For corespace 16 a little bit earlier, at 15% (Figure 5.31). Once again keeping in mind that task 13 is the most dominant task and not upgradable (very bad combination), it could be the reason GB has worse total task-list execution times than Versioning. This is also the reason RAlgA seems to be doing better with this task than the other algorithms, in contrast to what it happens with all the other tasks.

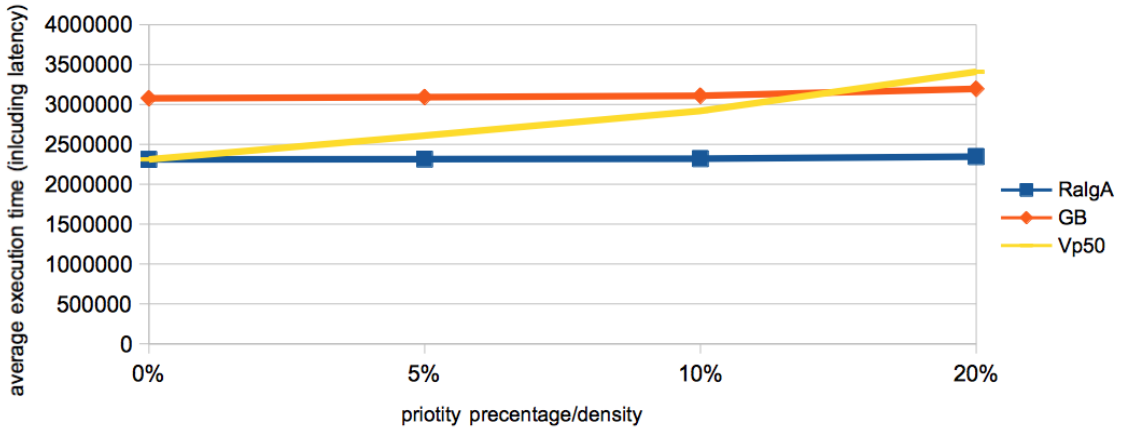


Figure 5.31: Task 13 average execution time (incl. latency) for each algorithm for corespace 8.

5.3 GB vs Versioning

5.3.1 GB vs GB++

With the introduction of priorities, GB got its final version: GB++. This version of GB has improved execution times (10% to the Versioning ones), forced priorities and upgrading by default. As shown in Figure 5.32, the algorithm seems to be doing worse in no priority/interrupt environment, but already with 5% priority it is doing better than GB in all corespace sizes and its performance is even enhanced as corespace and especially priorities/interrupts increase. There are hardly any systems without interrupts any more and priorities will probably be a popular feature, so there is no reason not to make GB++ the standard version of GB.

5.3.2 GB++ with Versioning execution times

Despite all its improvements GB has not managed yet to beat Versioning in total execution time of a long task list. Even with the improved execution times of 10% and even with priority density as 20% and corespace 16 it takes more time with GB scheduling for a specific task list to execute than with Versioning. The main reason is the overhead time of the generic binary compared to the optimal compilation of the Versioning binary. The name 10% times that [26] gives to those GB execution times is tricky, considering this is just the average of the individual speedups achieved with the improved compilation (Table 3.1). That is, however, no indication that GB should be only 10% worse

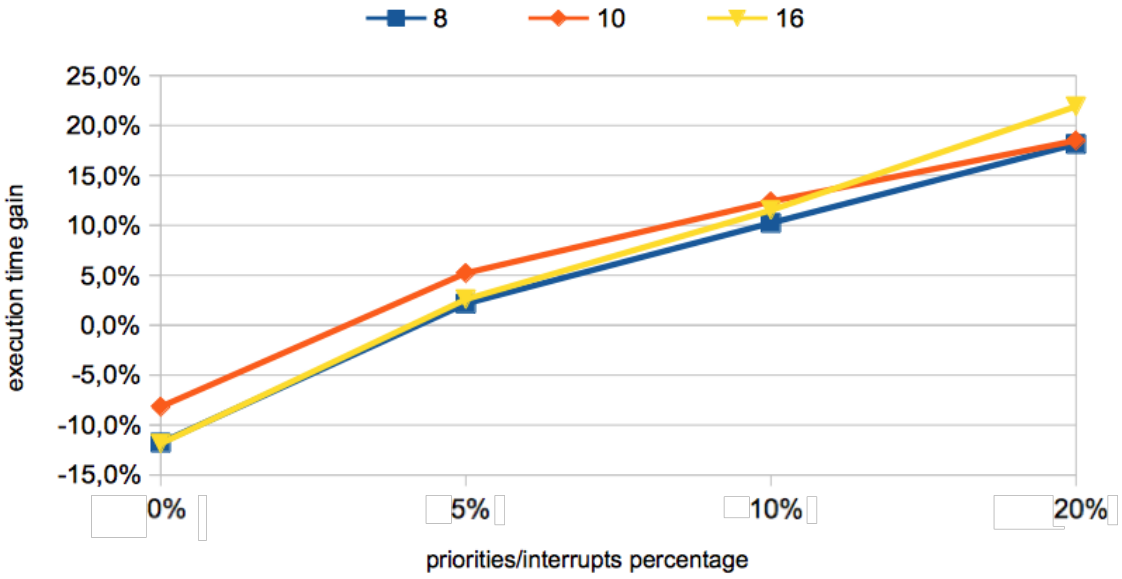


Figure 5.32: Speedup of GB++ to GB.

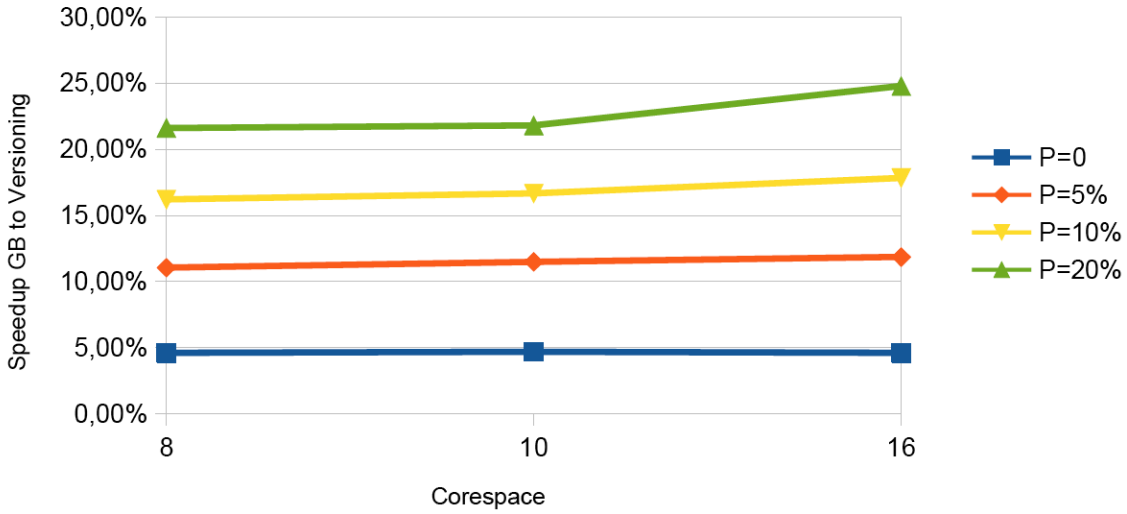


Figure 5.33: Running GB with Versioning execution times.

than Versioning. That average should have been a weighted average, since the tasks have great differences with each other. Having only 8% execution time overhead after compiling task 3 (a quite small task) cannot compensate for having 33% overhead for task 13, a task-giant.

The last version of the simulator has as its goal to detect how much the compilation has to improve so that GB reaches and exceeds Versioning's total execution times. It takes the Versioning execution times as input and schedules them with the GB algorithm several times (with the typical 3 corespace options and the typical 4 interrupt/priority density options), each-time making the times a little bit worse. It starts with exactly Versioning times and it ends with times 20% worse than them. For Versioning 50.000 cycles reloading penalty was used as usually.

GB always achieves better total execution time when the typical execution times of Versioning are used as an input (Figure 5.33). Confirming the previous results, GB's performance improves even further as priorities/interrupts increase and for corespace 16 there is even a small extra boost.

The Versioning execution times, however, are probably impossible to reach with GB compilation. We research therefore how much worse GB execution times can be and still GB be better than Versioning. The results are presented in Figure 5.34. Considering 0% interrupts is an unlikely situation and interrupt/priority percentage of 5% as a minimum, GB has chance to become as good or faster than Versioning if its execution times get on average no worse than 12% of those of Versioning. Since GB demands simpler hardware, does not demand 3 different binaries for each task being stored in the memory, supports task upgrading, copes better with interrupts and priorities and takes advantage of increased corespace better than Versioning, the benefits would be great.

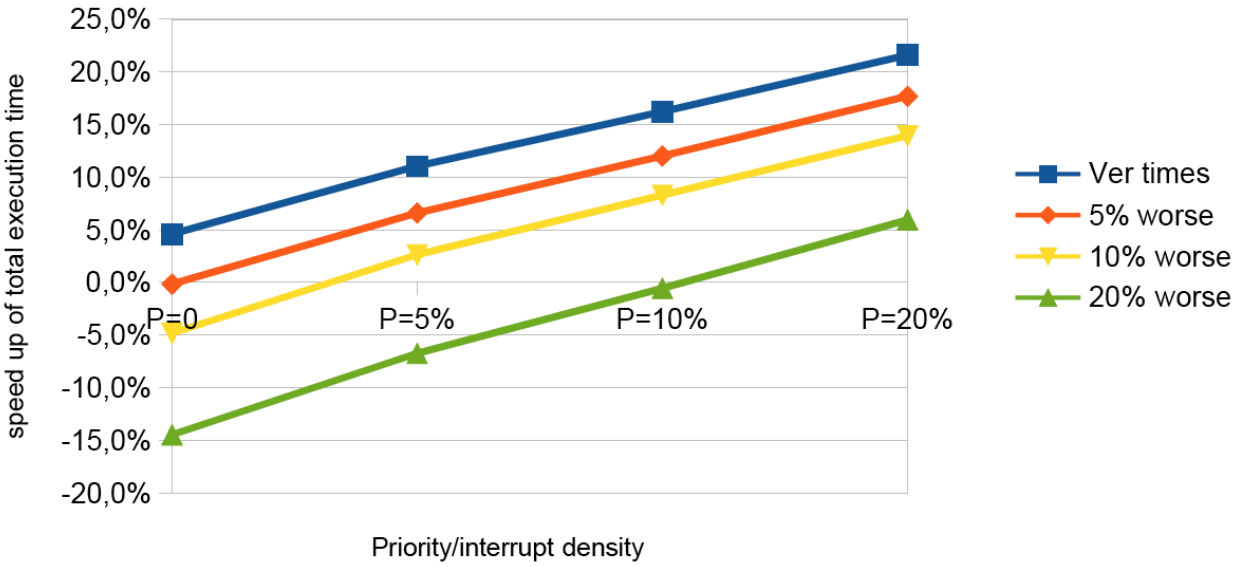


Figure 5.34: Speedup GB to Versioning for corespace 8.

5.4 Conclusion of Chapter 5

5.4.1 The upgrading paradox and 2D scheduling

We mentioned many times in this chapter the upgrading paradox. That is the phenomenon that despite the fact that each task is running faster on a bigger core, the whole task list runs slower. One would assume that 2 tasks would run faster as 4-issue one after the other rather than as 2-issue one next to each other.

Take task 1, adpcm, and task 11, qurt. They are both preferably 4 issue tasks. We will round up their execution times for the example to 30k and 20k for adpcm and 40k and 30k for qurt, or just 3, 2, 4 and 3. Running them next to each other as 2-issue would take $max(4, 3) = 4$ cycles whereas running them one after another as 4-issue would take $3 + 2 = 5$.

In all the research on ρ-VEX till now, running a task on a bigger core has been considered as a good thing because the task was being executed faster. But now that task is not the only one. There are also other tasks that compete for those resources too, so the question of "Is it better/faster to run each task as fast as possible one after another or is it better to let them run each a little bit slower but simultaneously next to each other?" To answer this question we have to start thinking two-dimensionally. Instead of taking into consideration only the execution time of a task, we should also take into consideration the amount of resources it takes. We can call this product of $ExecutionTime \times Resources$ as 'execution space' of the task.

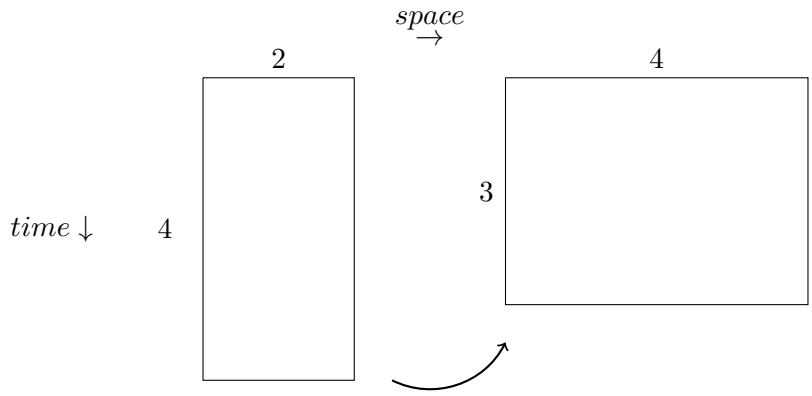


Figure 5.35: The execution space of qurt for 2-issue (left) and 4-issue (right).

Figure 5.35 depicts the core execution space of qurt in 2-issue and 4-issue, which is 8 and 12 accordingly. Assuming that the corespace is 4, running the task in 2-issue leaves execution space equal to 8 for another task to run in parallel. Running the task in 4-issue leaves execution space only 4. That is why when we bring in adpcm too (Figure 5.36) we see that it finishes later in 2-issue than in 4-issue. Qurt has benefited 1 cycle, but adpcm paid 2 for it.

Theoretically, when we upgrade, one dimension always doubles, so the other dimension has to halve if we want not to use more execution space than before upgrading. If we want to have an acceleration then we should even have more than 50% improvement in execution time (Figure 5.37). Which is by definition impossible, if one considers the way VLIW instructions work.

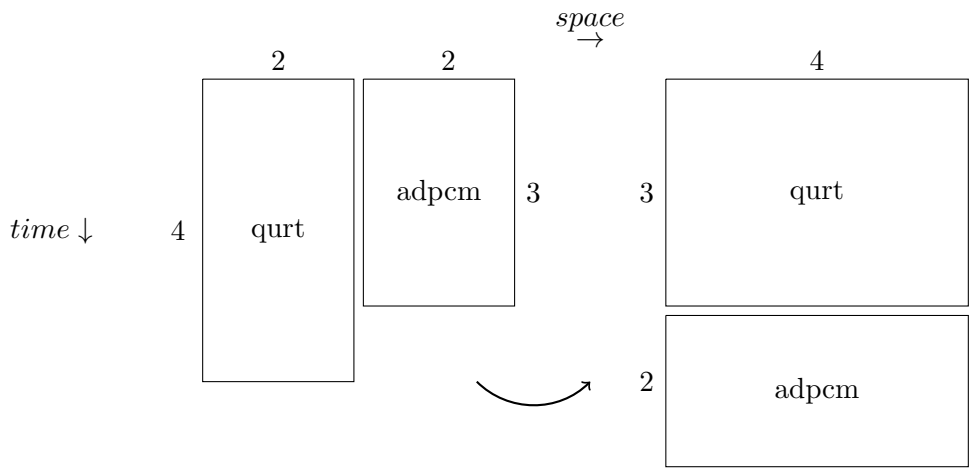


Figure 5.36: The execution space of qurt for 2-issue (left) and 4-issue (right).

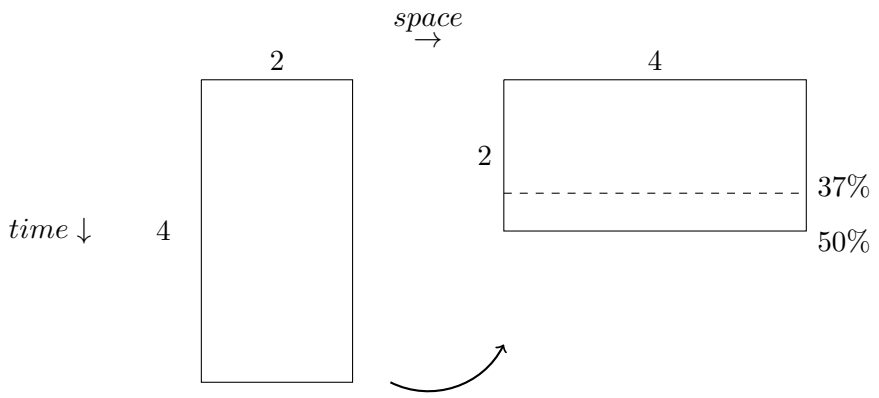


Figure 5.37: Theoretical prerequisite in order to have speedup by upgrading.

This is the reason why priorities cause slowdown to all algorithms. That does not mean that we should always run everything on 2-issue or do not support priorities. Without priorities the system would not be very interesting for real-time systems and some tasks can really handle some extra delay and for some other it is really important to finish a little bit earlier. Which algorithm does that better, for which tasks and for which not, we are going to see at the next section that studies task latency.

5.4.2 Summing up the results

In this chapter we presented the most important of the experiments that took place within this thesis. We calculated the utilisation of AlgA and RAlgA (Basic). We found that a task list window larger than 8 does not benefit the Basic algorithm. We showed that algorithms like AlgD and all-8 versions are achieving worse results, some of them even worse than RAlgA. We defined corespace 16 as a realistic upper limit for this research. We set a reloading penalty of 50.000 cycles for Versioning and discovered that it is still not as much as the downgrading penalty of GB and that its own downgrading penalty is even smaller, since most of its tasks are 2-issue.

We showed that GB++ copes better with priorities and interrupts than Versioning, but its bad execution times and the mono-dimensional task scheduling are keeping it behind Versioning. Along with priorities and interrupts came the notion of task latency. The most important conclusions from the latency study were:

- Small (execution time) and 2-issue tasks suffer more latency (%) than big tasks.
- GB benefits more from corespace increase than Versioning (as far as latency is concerned), especially on high priority density environments.
- Small tasks manage speedups with Versioning if they are 4-issue, with GB not, probably because of the fact that there are many more 4-issue and 8-issue tasks, whose priorities and upgrades increase even more the latency of small tasks.

- 8-issue tasks cannot make much use of their speedup, because they fall on other priority tasks. Their slowdown increases with the increase of the priority density and decreases with the increase of corespace.
- In tasks that GB can make use of its upgrading ability, it is the fastest algorithm.
- GB is an algorithm where we can be generous with priorities (which makes it a good candidate for multi-level priorities) in contrast to Versioning. On top of that, in high levels of priority density corespace increase improves its performance. Both qualities are following the trends of technology.

Enhancing GB with all those tools (forced priorities, default upgrading, etc) lead to the best GB version of all: GB++. This version not only performs better than Versioning if they had the same typical execution times, but even if GB times were 12% worse than those of Versioning.

6 Conclusions

6.1 Summary

After getting an inside look of the Processing component of the ERA platform and understanding the particularities of it, as well as of its main building block, the ρ-VEX core, we tried to find in the bibliography an available scheduling algorithm to implement as the Hardware scheduler of ERA. This literature research did not yield any results, both because of the complexity of the problem, as well as the pioneer characteristics of ERA that we would like to take advantage of. Thus, we tried to design simple scheduling algorithms, especially tailored for the ERA platform and see how they perform.

We presented the several algorithms that were developed and tested during the thesis, starting from a very Basic task scheduling algorithm and moving to algorithms more specific to the ERA platform. Such as Versioning, which downgrades the task by scheduling it to run on a smaller core to avoid creating bubbles that Basic creates. Running on a smaller core means bringing from the memory a different binary of the task, compiled especially for that core. That takes time, which is called the 'reloading penalty'. To deal with the reloading penalty Generic Binary was invented creating a binary of a task that can run on any core. That also created the ability to downgrade tasks on-the-fly to make space for new tasks, support priorities and handle interrupts without big delays for the tasks already running. All these lead to a new algorithm, AlgB or GB. But it also brought the idea of upgrading a task when resources become available, instead of bringing immediately a new task for execution. That is GB++. Some very unsuccessful algorithms like AlgBall8 and AlgD were also developed.

The most important of the experiments that took place within this thesis were presented in Chapter 5. The utilisation of AlgA and RAlgA (Basic) was calculated. We found that a task list window larger than 8 does not benefit the Basic algorithm. We showed that algorithms like AlgD and all-8 versions are achieving worse results, some of them even worse than RAlgA. We defined corespace 16 as a realistic upper limit for this research. We set a reloading penalty of 50.000 cycles for Versioning and discovered that it is still not as much as the downgrading penalty of GB and that its own downgrading penalty is even smaller, since most of its tasks are 2-issue.

We showed that GB++ copes better with priorities and interrupts than Versioning, but its bad execution times and the mono-dimensional task scheduling are keeping it behind Versioning. Along with priorities and interrupts came the notion of task latency. The most important conclusions from the latency study were:

- Small (execution time) and 2-issue tasks suffer more latency (%) than big tasks.
- GB benefits more from corespace increase than Versioning (as far as latency is concerned), especially on high priority density environments.
- Small tasks manage speedups with Versioning if they are 4-issue, with GB not, probably because of the fact that in GB there are many more 4-issue and 8-issue tasks, whose priorities and upgrades increase even more the latency of small tasks.
- 8-issue tasks cannot make much use of their speedup, because they fall on other priority tasks. Their slowdown increases with the increase of the priority density and decreases with the increase of corespace.
- In tasks that GB can make use of its upgrading ability, it is the fastest algorithm.
- GB is an algorithm where we can be generous with priorities (which makes it a good candidate for multi-level priorities) in contrast to Versioning. On top of that, in high levels of priority density corespace increase improves its performance. Both qualities are following the trends of technology.

Enhancing GB with all those tools (forced priorities, default upgrading, etc) lead to the best GB version of all: GB++. This version not only performs better than Versioning if they had the same typical execution times, but even if GB times were 12% worse than those of Versioning.

6.2 Main problem statement & contributions

The goal of this thesis was to find available or possible scheduling algorithms for the ERA Multicore ρ-VEX Processor, test them and analyse the results.

- We delivered an open-source parametrised simulator of the Task scheduler of ERA, on which future research can be performed on different test-cases, as well as development and testing of new task scheduling algorithms.
- This thesis showed that Generic Binary, though not the fastest algorithm, yet it is the most promising one for now and the future, because it is scalable to corespace, tolerant to interrupts and it has potential to be used in Real-time systems too.
- Versioning however was found to be the fastest algorithm in terms of total execution time of a long task list.
- We found the minimum standard (exec. times maximum 12% worse than Versioning) that the Generic Binary has to comply with in order to be able to compete with Versioning in terms of total execution time.

- We showed that thinking about execution time only is not enough when we move from linear to parallel execution in platforms like ERA and that we should start thinking about two-dimensional scheduling, if the resources allow the luxury of a more sophisticated algorithm.
- We finally showed that parallel execution is much more complicated than linear execution and introducing characteristics that help on one aspect (task execution speedup) can bring the opposite results somewhere else (slowdown of non-priority tasks).

6.3 Future work

As proposal for future research I would recommend that the first thing that happens is that the algorithms are implemented in hardware and tested in a real ERA setup. Further, a more fair (for each task) and more accurate definition of the Versioning reloading penalty would be beneficial. Finally, there should be some research on the possibility of 2D task scheduling, that is scheduling according to the execution space, not the execution time of a task.

Bibliography

[1] S. Wong, A.A.C. Brandon, F. Anjam, R.A.E. Seedorf, R. Giorgi, N. Puzovic, S. McKee, L. Carro, and G. Keramidas. Early Results from ERA – Embedded Reconfigurable Architectures. In Int. Conference on Industrial Informatics (INDIN), pages 816-822, Lisbon, Portugal, July 2011.

[2] S. Wong and F. Anjam. The Delft Reconfigurable VLIW Processor. In International Conference on Advanced Computing and Communications (ADCOM), pages 242-251, 2009.

[3] S. Wong, T. van As, and G. Brown. ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor, in Proceedings IEEE International Conference on Field-Programmable Technologies (ICFPT08), Dec 2008, pp. 369-372.

[4] J. Fisher, P. Faraboschi, and C. Young. Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann, 2004.

[5] R.A.E. Seedorf, F. Anjam, A.A.C. Brandon, and S. Wong. Design of a Pipelined and Parameterized VLIW Processor: ρ-VEX v.2.0. In 6th HiPEAC Workshop on Reconfigurable Computing (WRC), 2012.

[6] F. Anjam, S. Wong, and M.F. Nadeem. A shared Reconfigurable VLIW Multiprocessor System. In International Parallel and Distributed Processing Symposium (IPDPS-RAW), pages 1-8, 2010.

[7] Fakhar Anjam. Run-time Adaptable VLIW Processors. PhD Thesis, TU Delft, 2013.

[8] F. Anjam and S. Wong. Configurable Fault-Tolerance for a Configurable VLIW Processor. In International Symposium on Applied Reconfigurable Computing (ARC), pages 167-178, 2013.

[9] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2006.

[10] Henk Corporaal. “Microprocessor architectures from VLIW to TTA”, Wiley, 1998.

[11] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood.“Lx: A Technology Platform for Customizable VLIW Embedded Processing”, in Proceedings of the 27th annual International Symposium of Computer Architecture (ISCA 00), June 2000, pp. 203 - 213.

[12] Anath Grama, Anshul Gupta, George Karypis, Vipin Kumar. Introduction to Parallel Computing, 2nd edition, 2002.

[13] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, CA, 1999.

[14] Keqin Li. Experimental Performance Evaluation of Job Scheduling and Processor Allocation Algorithms for Grid Computing on Metacomputers. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS-04).

[15] Z. Xu, X. Hou and J. Sun, "Ant Algorithm-Based Task Scheduling in Grid Computing", Electrical and Computer Engineering, IEEE CCECE 2003, Canadian Conference, 2003.

[16] Y. Zhang, C. Koelbel, and K. Kennedy, "Relative Performance of Scheduling Algorithms in Grid Environments", Proc. IEEE Seventh Int'l Conf. Cluster Computing and the Grid, pp. 521-528, May 2007.

[17] H. Youness, et al. Efficient partitioning technique on multiple cores based on optimal scheduling and mapping algorithm, Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), pp. 3729-3732, 2010.

[18] Yi Pang, Lifeng Sun, Jiangtao Wen, Fengyan Zhang, WeiDong Hu,Wei Feng, Shiqiang Yang, "A Framework for Heuristic Scheduling for Parallel Processing on Multicore Architecture: A Case Study With Multiview Video Coding" IEEE Trans. Circuits Syst. Video Techn. vol. 19, pp.1658-1666, Nov. 2009.

[19] Xiaozhong Geng, Gaochao Xu, Dan Wang, Ying Shi. "A Task Scheduling Algorithm Based on Multi-Core Processors", International Conference on Mechatronic Science. Electric Engineering and Computer (MEC), 2011.

[20] R. Hoffmann, A. Prell, and T. Rauber. Dynamic task scheduling and load balancing on cell processors. In Parallel, Distributed and Network- Based Processing (PDP), 2010 18th Euromicro International Conference on, pages 205-212, 2010.

[21] H. Tang, K. Rupnow, P. Ramanathan and K. Compton, "Dynamic binding and scheduling of firm-deadline tasks on heterogeneous compute resources," Proc. in RTCSA, 2010, pp. 275-280.

[22] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux and H.-U. Heiss. Evaluating thread placement based on memory access patterns for multi-core processors. In Proc. of HPCC'10, 2010.

[23] H. Topcuoglu, S. Hariri, and M. Wu, "Task Scheduling Algorithms for Heterogeneous Processors". Proc. Heterogeneous Computing Workshop, pp. 3-15, Apr. 1999.

[24] A.P.D. Binotto, C.E. Pereira, A. Kuijper, A. Stork, and D.W. Fellner, "An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms", in IEEE 13th International Conference on High Performance Computing and Communications (HPCC), Sept. 2011, pp. 78-85.

[25] H.-K. Tang, P. Ramanathan, and K. Compton. "Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources". In International Conference on Parallel Processing (ICPP), 2011, pp. 753-762.

[26] A. Brandon and S. Wong. Support for Dynamic Issue Width in VLIW Processors using Generic Binaries. In Design, Automation, and Test in Europe Conference (DATE), pages 827-832, 2013.

[27] F. Anjam, Q. Kong, R.A.E. Seedorf, and S.Wong. A Run-time Task Migration Scheme for an Adjustable Issue-slots Multi-core Processor. In International Symposium on Applied Reconfigurable Computing (ARC), pp. 102?113, 2012.

[28] F. Anjam, M. Nadeem, and S.Wong. Targeting Code Diversity with Run-time Adjustable Issue-slots in a Chip Multiprocessor. In Design, Automation and Test in Europe Conference (DATE), pp. 1358-1363, 2011.

Appendix A

In this appendix some early research results are presented. Figure A.1 depicts some early results from long-run test. For example the comparison of the total execution time of a long list of Algorithm A (AlgA) to AlgB and AlgB to AlgBall8.

Surprisingly we find that GB is slower than AlgA. That happens because at this point the algorithm did not have a window to limit it. So practically it could search to infinite for a task that fits. This way AlgA was making a perfect allocations of all tasks, almost without any bubbles. This very early version of GB cannot overcome the fact that its average execution times suffer a big overload compared to the ideally compiled versions of AlgA/Versioning binaries.

An effort to tackle that with setting the biggest core size (8) as preferred for all tasks also fails. Specifically AlgBall8 takes twice the time to run than AlgB. That was an early sign that, as we will see later on, at least in terms of total execution time of the task list, we actually do not benefit from running tasks on larger cores, since we do not run it two times faster but slower than that. A small proof of that is also the fact that the algorithm is doing better for corespace 10 rather than for corespaces multiples of 4 or 8. Corespace is included on purpose in the tests in order to investigate fault-tolerance (situations where one or more cores become defect) of the algorithms and generally investigate the behavior of the algorithms in non-symmetric set-ups.

But the GB and Versioning are doing better to Realistic Algorithm A (RAlgA) as

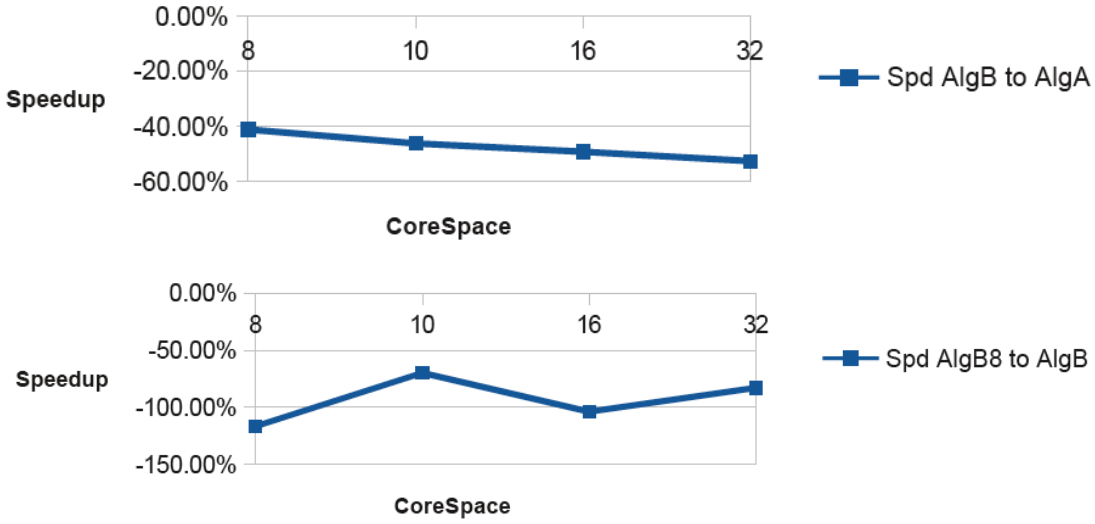


Figure A.1: Some early test results actually showed slowdown instead of speedup.

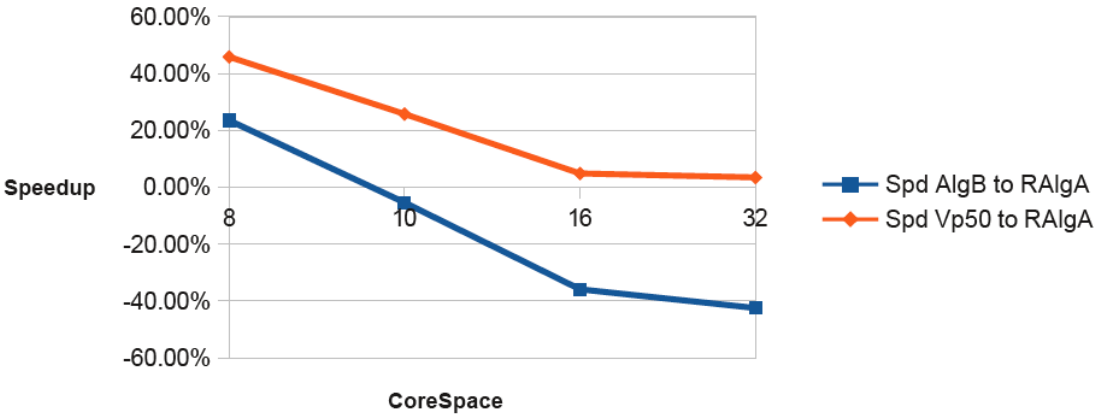


Figure A.2: Some early test results of the total execution time speedup of GB and Versioning (penalty=50) to RAlgA.

shown in Figure A.2. The benefit disappears as the corespace increases. That happens because the Basic Algorithm finds more and more space to accommodate bigger tasks, so less bubbles are created, whereas the other algorithms are more specialised in dealing with lack of space. The disappointing performance of GB is a result of the execution times. This group of execution times is on average 40% worse than Basic/Versioning times and therefore is referred as 'GB 40% times'. Later on better binaries were announced by [26] after improved compilation, which are on average 10% worse than Basic/Versioning for 4-issue cores and about 20% for 2-issue cores, so it is referred as 'GB 10% times'.

Those preliminary tests also show little influence of the reloading penalty for Version-

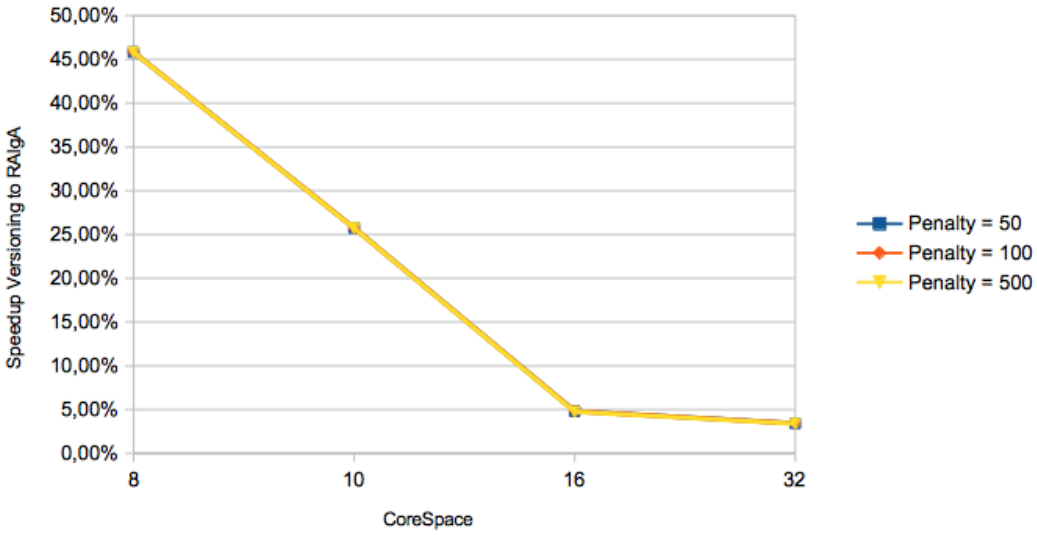


Figure A.3: Early arbitrary Versioning penalty tests.

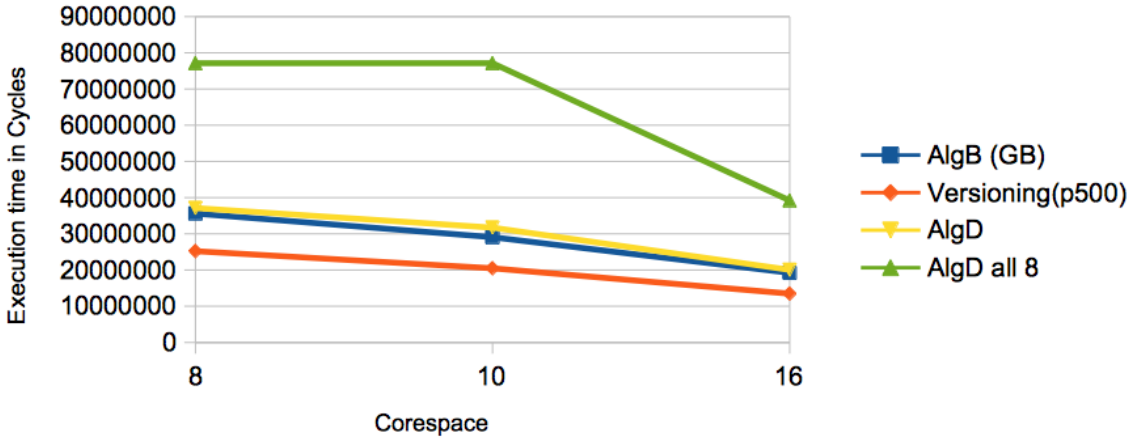


Figure A.4: Early results from AlgD: total task list execution time.

ing. However these values (50 to 500) are arbitrary. In the following tests the value of the penalty is more realistic and is expected to be much higher. The corespace definitely contributes more crucially to the drop for the Versioning speedup to Basic (Figure A.3), since, as mentioned before, more corespace causes less bubbles in Basic.

Another result from those early tests was the performance of AlgD. AlgD (a mix of AlgB and AlgA) was invented to tackle the problem of too much downgrading. As seen in Figure A.4 (total task-list execution time in cycles) and Figure A.5 (speedup to RAlgA) AlgD is doing slightly worse than Generic Binary (GB). The All-8 versions of AlgD and GB even have a slowdown instead of speedup (as shown in Figure A.5).

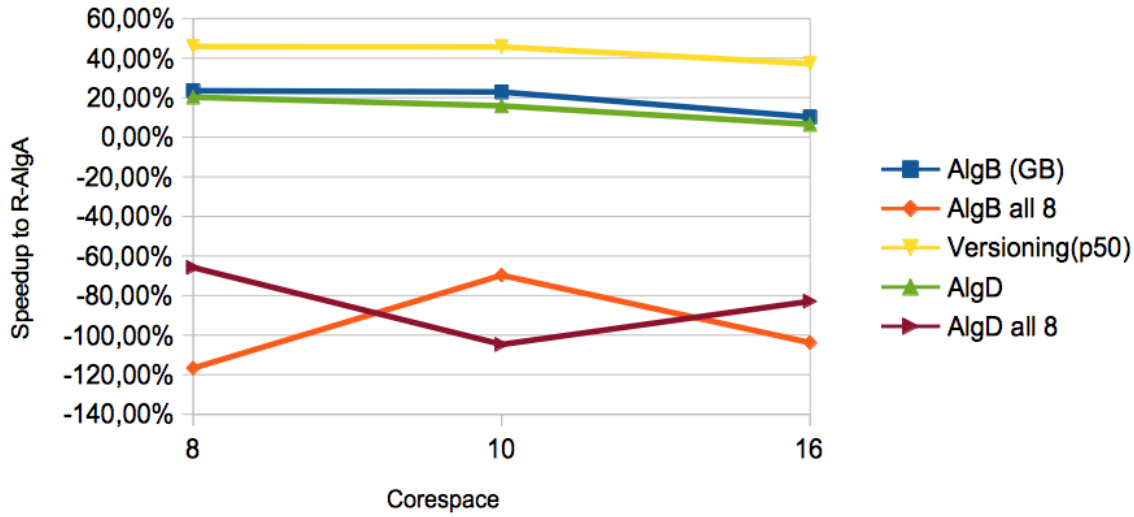


Figure A.5: Early results from AlgD: speedup to RAlgA.