

Performance Evaluation of the Nios Architecture

Ivan Matosevic Franjo Plavec

Department of Electrical and Computer Engineering
University of Toronto

Abstract

In this paper we study the performance of the Nios architecture from Altera Corporation. Nios is a soft-core processor described in a Hardware Description Language (HDL), targeting Altera Field Programmable Gate Array (FPGA) devices for the implementation. We base our study on the UT Nios implementation of the Nios architecture developed at the University of Toronto. We explore how different pipeline organizations and other architectural parameters influence the performance of the UT Nios implementation, by using a simulation based methodology. For this purpose we developed a SimpleScalar model of the 32-bit Nios architecture, based on an existing model for the 16-bit Nios architecture. Our results indicate that the best performance is achieved using 4-stage pipelined UT Nios.

1 Introduction

Designing a new computer architecture is a challenging task. An architect has to choose among many possible options for various architectural parameters to meet one or more goals for a given set of target applications. One of the main goals is often the performance defined through the execution time of typical programs that will be running on the architecture. Most modern processor architectures have pipelined datapath for improved performance. The organization of the pipeline has a large impact on the processor performance. While deeper pipeline decreases logic delays in the design, and thus increases the maximal operational frequency (F_{\max}), it also incurs more pipeline stalls due to various pipeline hazards [1].

In this work we study how different pipeline organizations influence the performance of the Nios architecture. Nios is a commercial soft-core processor from Altera Corporation [2]. Nios is a RISC architecture intended for use in the embedded applications. For the purpose of this study we use UT Nios, an implementation of Nios architecture developed at the University of Toronto [3]. Although the soft-core processors are easy to modify, compared with their counterparts implemented in ASICs, getting a fully functional processor is still time consuming, because it requires a lot of testing and debugging. More specifically, different pipeline

organizations incur different hazards, which may be hard to comprehend and account for. Therefore, the methodology that would give good performance estimates in the shorter time is desired. In this paper we propose a simulation based methodology for the performance estimation.

We estimate the performance of various pipeline organizations by estimating its F_{\max} and the number of instructions executed per clock cycle (IPC). F_{\max} is estimated by reorganizing the existing datapath, synthesizing the new design, and performing the timing analysis on the design. IPC is estimated using the developed SimpleScalar based simulator to run a subset of benchmarks from the UT Nios benchmark set [3]. Combining these results gives a good estimate of the wall clock execution time of the programs in the benchmark set, which serves as a basis for the performance comparison. Aside from the pipeline organization, we also consider how the choice of FPGAs as a target technology limits the possible choices in processor design, and how these limitations influence the performance. We find that a 4-stage pipeline with fully implemented data forwarding performs the best over the set of benchmarks used.

The rest of the paper is organized as follows. The following section gives an overview of the Nios architecture. Section 3 presents the UT Nios implementation of the Nios architecture. Our experimental methodology is described in Section 4, and the results of our experiments are presented in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2 Nios Architecture

Nios is a RISC processor, and in the original implementation from Altera it is implemented as a 5-stage pipelined design. Since Nios targets FPGAs for the implementation, it is highly customizable. This allows users to better tailor the system to the needs of the target application. Customizable parameters include the size of the on-chip memory, a selection of various peripherals implemented on-chip, and the parameters of the processor itself. The processor can be configured to use 16- or 32-bit datapath [4,5]. Some optional instructions can be omitted if they are not needed. The processor can be optimized for speed or area savings. The sizes of the instruction and data caches can be customized for the 32-bit datapath. Custom

instructions can be added to further improve the performance of the target applications [6].

Nios processor features a large windowed general-purpose register file. The size of the register file can be selected among the three predefined values (128, 256, 512), but only a window of 32 registers is visible to programs at any given time. The visible register window is defined by the current windows pointer (CWP) field in the control register set. The window is changed using special instructions SAVE and RESTORE. The register windows overlap, which enables fast parameter passing between procedures. The compiler provided with Nios is based on the *gcc* compiler from the GNUPro Toolkit. It supports the option to generate the code that does not use the register window capabilities [7].

Nios instruction set is optimized for embedded applications. Since the available memory in the embedded systems is limited, all instructions are 16 bits wide, which typically results in reduced code size compared to the architectures with 32-bit instruction word [1]. Nios uses the two-operand instruction format, and several addressing modes. Since 16-bit instruction word severely limits the size of the immediate operands, special 11-bit register (K register) is used to form larger operands. It is preloaded with an immediate value using PFX instruction, and the value in the register can be used by the instruction following the PFX. All branches and jumps are unconditional, and conditional execution is achieved using conditional skip instructions, which skip the following instruction subject to a given condition. A skip instruction followed by a branch or a jump performs conditional branch or jump operation. Most branch instructions have the delay slot behaviour with the delay slot of one instruction. The Nios instruction set has only simple logic and arithmetic instructions that support only integer operations.

Altera's implementation of the Nios architecture (Altera Nios) is a 5-stage single-issue pipeline with separate data and instruction masters. Most instructions will take 5 cycles to execute, resulting in a throughput of 1 instruction per cycle when there are no pipeline stalls. The exceptions are control-flow instructions, whose latency depends on the latency of the instruction memory; loads and stores, whose latency depends on the latency of the data memory; MUL instruction (performs integer multiplication), whose latency depends on the target FPGA device family; and shift instructions, which take 6 cycles to execute.

GNUPro Toolkit provided with Nios generates a custom software development kit (SDK) for each system developed. The SDK contains library routines and drivers needed for the system software development.

Since many details on the organization of the Nios from Altera are not provided in the Altera documentation, and the implementation itself is protected by a license, we

have based our study on the UT Nios implementation, which is described in the next section.

3 UT Nios Implementation

UT Nios is an implementation of the Nios architecture with the functionality equivalent to that of the Altera Nios, except for some optional components. The implementation described in [3] is a 4-stage pipeline presented in Figure 1. The four stages are fetch (F), decode (D), operand (O) and execute (X), as marked at the top of the figure.

In the fetch stage, instructions are fetched from the instruction memory, and forwarded to the decode stage. In the decode stage instruction is decoded, and its operands are read from the general-purpose register file. In the operand stage, operands are selected among possible sources. Since the Nios architecture supports many addressing modes, the operands may be coming from the register file or immediate operands. Furthermore, the immediate operands may be formed by using the value in the K register or sign-extended from the immediate operands of various possible lengths specified in the instruction word. If the instruction in the operand stage uses the result of the instruction in the execute stage, data forwarding is also performed in the operand stage. In the execute stage instructions are executed and the results are committed to the register file and control registers. The only exceptions are the control-flow instructions and the PFX instruction, which commit in the operand stage. Intermediate results of the instruction execution are stored in the pipeline registers IR, D/O, and O/X.

Most instructions execute in five clock cycles, with the exception of the control-flow instructions and the memory operations. The control-flow instructions generally incur at least two cycles of branch penalty, so that they take at least 7 clock cycles. The exact number of cycles needed for these instructions depends on the instruction memory latency.

In our experiments we consider different pipeline configurations, some of which partition the datapath into pipeline stages differently than the original UT Nios implementation. Therefore, we split the decode stage D into D1 and D2, and execute stage X into EX and WB (write-back) partitions, as marked in the bottom of Figure 1.

Several parameters of the UT Nios implementation can be customized, including general-purpose register file size and the choice to implement the instruction decoder in on-chip memory or in logic. Implementing decoder in the on-chip memory causes the data from the instruction decoder module to become available one clock cycle after the address signals are set. The general-purpose register file, which is also implemented in memory, experiences similar behaviour. In the original UT Nios implementation this is compensated for by setting the memory address a cycle before the data is actually needed. In case of the instruction decoder, this is done by forwarding the instruction from the

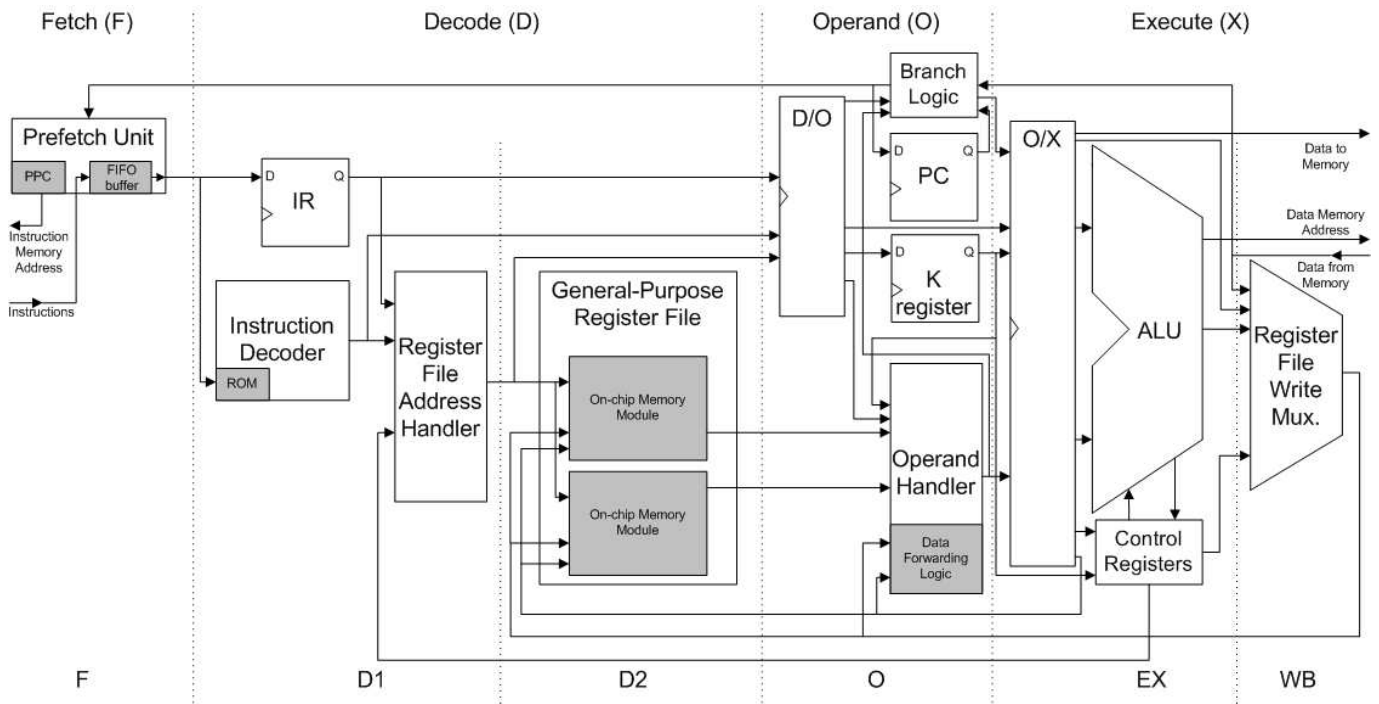


Figure 1 UT Nios datapath [3]

fetch stage directly to the instruction decoder, without first going through the instruction register (IR). Output of the general-purpose register file, on the other hand, is not buffered in the D/O pipeline registers, but forwarded directly to the operand stage. This compensates for the synchronous operation of the memory implementing the register file and saves the logic resources for the D/O pipeline register. Using on-chip memory for the implementation of the instruction decoder and the general-purpose register file limits the range of possibilities for the pipeline organization. For instance, it is not possible to perform the instruction decoding and the operand reading in the same clock cycle. The Nios architecture itself also limits the range of possibilities for the pipeline organization. Many supported addressing modes, coupled with the register window support, require the instruction to be decoded before its operands can be read, because some instructions use implicit register operands. Therefore, registers cannot be read in parallel with the instruction decoding like in traditional RISC architectures [1]. The detailed description of the UT Nios implementation can be found in [3].

In our experiments we explore how the described limitations influence the performance of various pipeline organizations. The next section presents our experimental methodology.

4 Experimental Methodology

In this section, we present the pipeline organizations whose performance was evaluated in this work. We also present the methodology used to estimate the performance of the pipeline organizations in terms of both the IPC and the clock cycle period.

4.1 Pipeline organizations

To explore the performance of various pipeline organizations, and in turn of the Nios architecture, we consider two sets of possible organizations.

In the first set, we only vary the number of stages, and the level of data forwarding in the existing UT Nios configuration. The first set is partially a review of the work presented in [3]. During the development process of the UT Nios, the simulator was not available, and the decisions were often based on estimates of the performance of typical applications for typical RISC architectures. Therefore, this work reviews these decisions in more formal fashion.

In this set, we explore the performance of seven different pipeline organizations:

Stages	Register File Implementation	Comment	Associated penalties (cycles)				Clock cycle (ns)	F _{max} (MHz)
			Branch	Memory	Data	S/R		
2	Memory	Instruction Decoder in Logic		1			22.94	43.6
3	Memory		1	1			22.85	43.8
4	Memory		2	1			11.87	84.3
4	Memory	No Data Forwarding	2	1	1		11.88	84.2
5	Memory		2	1	1		11.50	86.9
5	Memory	No Data Forwarding	2	1	2		10.94	91.4
5	Memory	Best F _{max}	2	2	1	1	10.31	97.0
2	Logic			1			24.91	40.1
3	Logic		2	1			13.98	71.5
4	Logic		3	1			11.36	88.0
5	Logic		3	1			11.00	90.9

Table 1 Pipeline organizations

- 2-stage pipeline with one stage consisting of the F and D1 partitions, and the other stage consisting of D2, O, EX and WB partitions. To enable the proper operation of the register file implemented in memory, we implement the instruction decoder in logic, and place it in the first stage together with the fetching of the instructions. We remove the D/O and O/X pipeline registers to join the last four partitions into a single stage.
- 3-stage pipeline with stages F, D, and the last stage consisting of O, EX and WB partitions. To achieve this, O/X pipeline register was removed from the UT Nios.
- 4-stage pipeline with the original four stages described in the previous section and used in [3].
- 4-stage pipeline equivalent to the one previously described, but without the data forwarding logic. Removing the data forwarding logic reduces the amount of logic used for the processor implementation, and reduces the critical path if the data forwarding logic is a part of it.
- 5-stage pipeline similar to the 4-stage, except that the X state was split in two stages, EX and WB, by introducing additional pipeline registers between the two partitions.
- 5-stage pipeline without the data forwarding logic.
- 5-stage pipeline with the data forwarding logic, memory data register (MDR), and registered path between the CWP field in the control register set and the register file address handler.

These pipeline organizations were derived from the simple organization with only two stages by identifying critical paths. Pipeline stages were then introduced to remove these critical paths. The last pipeline organization is the configuration with the best possible F_{max} that could be achieved by simple pipelining. The critical paths in this

design include control logic, which is hard to pipeline, and it is unlikely that further pipelining would result in the improved performance. We also do not consider a non-pipelined design (only one stage) for two reasons. First, a design using the general-purpose register file in the on-chip memory has to be pipelined because of the synchronous operation of this memory. Second, since the instruction memory is also synchronous, the instruction master has a latency of one cycle. Therefore, in a non-pipelined design, each time a new instruction read is issued, it would take one cycle for the instruction to be read. Hence, this configuration would perform poorly both in terms of the IPC and in terms of the clock cycle period, because the period could only increase if the IR pipeline register was removed.

Memory data register was introduced to remove the critical path through the data bus going to the data memory. It could be seen as an additional pipeline stage (memory), although it is not a pipeline stage in the traditional sense, because the instructions that do not access the memory proceed directly from the EX stage to the WB stage, without having to pass through the memory “stage”. The path from the CWP field in the control register set to the register file address handler was identified as critical. Whenever the SAVE or RESTORE instruction is issued to change the currently visible register window, the new value of the CWP is forwarded to the register file even before the SAVE or RESTORE instruction commits. This is necessary because the instructions following the SAVE or RESTORE instruction may need the new value of the CWP to read the correct registers. Registering this path removes the critical path, but introduces one stall cycle for every SAVE and RESTORE instruction.

Because the general-purpose register file implemented in memory limits the possibilities for the pipeline organization, we also explore how implementing a smaller register file in logic influences the performance. [3] suggests that many applications do not suffer significant performance penalty

when the register window capabilities of the Nios architecture are not used. Therefore, we explore how implementing a small register file of only 32 registers (equal to the size of a register window in the original register file) influences the performance, and more importantly, how it influences the possibilities for the pipeline organizations. We consider the following pipeline organizations:

- 2-stage pipeline, where one stage consists of the F partition and the other stage consists of D1, D2, O, EX and WB partitions.
- 3-stage pipeline, with the first two stages consisting of F and D partitions, and the third stage of O, EX, and WB partition.
- 4-stage pipeline with the first three stages corresponding to the partitions F, D1, D2, and the last stage consisting of partitions O, EX, and WB.
- 5-stage pipeline equivalent to the 4-stage organization, except for the last stage, which is split into two stages, one containing only the O partition, and the other containing EX and WB partitions.

These pipeline organizations were selected using the same methodology described for the first set of pipeline organizations. It is interesting to notice that the methodology resulted in different pipeline organizations. The reason for that is the implementation of the general-purpose register file in logic. Since the logic elements (LEs) in FPGAs typically contain only a single register per LE [8], the register file implementation in memory uses many LEs (around 1700). Since in the FPGAs the routing dominates the combinational delays [9], and a large register file requires lots of routing resources, it becomes beneficial to divide the original decode stage into two stages to reduce the critical path through the register file. Such division does not bring significant benefits when the register file is implemented in memory, because the inputs to the on-chip memory are already registered. We did not consider the pipeline organizations without the data forwarding logic, because those organizations demonstrated poor performance for the first set of pipeline organizations. Furthermore, the logic savings obtained by removing the data forwarding logic are negligible compared to the logic consumed by the register file implemented in logic, which suggests that this configuration would never be used in area-critical applications.

All described pipeline organizations with the pipeline stalls associated with each organization are presented in Table 1. The table shows the number of various pipeline stalls and the associated penalties. Branches cause stalls because the instructions following the branch are fetched before the branch actually commits, so these instructions have to be flushed from the pipeline, effectively causing pipeline stalls. The branch penalty is equal to the distance between the fetch stage and the stage in which the

branches commit. The only exceptions are pipeline organizations with high clock period, in which case we assume that the branch commits on a negative clock edge (in the middle of the cycle), which reduces the number of branch stalls by one, as suggested in [3].

Memory stalls are caused by the latency of the data memory. They cause one stall cycle per every memory operation, except when the memory data register is used, which introduces an additional stall cycle. Data hazards are only present when the data forwarding is not implemented, or the write-back occurs later in the pipeline than the execution. Each of the two factors accounts for one stall cycle. S/R label in Table 1 stands for the SAVE and RESTORE instructions, which cause one stall cycle when there is a register on the path from the CWP field in the control register set to the register file address handler.

The timing information; the duration of the clock cycle, and the corresponding maximal operational frequency, presented in the table were obtained by performing the timing analysis on the synthesized design. The methodology used in obtaining the timing estimates is presented in the next section.

4.2 Timing

The pipeline organizations presented in the previous section partition the datapath into different numbers of pipeline stages. To get precise timing estimates, the datapath has to be connected with the control unit of the processor, and the processor itself has to be connected to other components in the system.

If the control unit could be implemented efficiently, we would have a complete design that could be downloaded into an FPGA, and whose performance could be tested on the actual hardware. However, implementing a control unit for each of the pipeline configurations is time-consuming. Therefore, we use the control unit of the original 4-stage pipelined UT Nios for all pipeline organizations. Since the control unit tends to be more complex for deeper pipelines, we can assume that the timing estimates obtained using this methodology is overly pessimistic for the pipelines with the depth less than 4, and overly optimistic for pipelines with the depth greater than 4. Since the majority of the delays are contained in the datapath, we believe that the errors in the estimates are small.

To connect the UT Nios to the rest of the system, we assume the minimal system with only two on-chip memory modules, used as the instruction and data memories. Both instruction and data memory modules are 16 Kbytes large. Instruction master of the UT Nios is connected only to the instruction memory, while the data master is connected only to the data memory.

We synthesize our designs using the synthesis tool integrated in the Quartus II CAD tool. We use Quartus II version 3.0, and target the EP1S10F780C6ES Stratix FPGA

device [8]. We use the Quartus II Timing Analyzer to obtain the maximal operational frequency, and to analyze the critical paths in the design. This helps us understand which parts of the datapath would benefit the most from introducing additional pipeline stages.

Previous work [3] has found that the seed number, which defines the initial placement of the design in a target device used by a placement and routing algorithm in the CAD design flow, can have significant influence on the final synthesis results. Therefore, we synthesize the design 10 times, each time with a different seed, and use the best result obtained. The results of the timing analysis of all pipeline organizations considered are given in Table 1.

4.3 Simulation

The configurations of the UT Nios architectures were tested using an execution-driven, cycle-accurate simulator based on the SimpleScalar tool set [11]. In the course of the UT Nios project, a simulator based on the SimpleScalar was developed for the 16-bit Nios architecture [12]. We have modified this simulator to support the 32-bit architecture. For each tested pipeline configuration, we have developed a version of the simulator that finds the exact number of stall cycles incurred by each type of the control and data hazards. The simulated architecture configuration can be changed using a *#define* directive in the source code of the compiler. The compiler was run on a PC running Windows 2000, using a Cygwin shell.

For the performance evaluation, we used a subset of the UT Nios benchmark set [3]. The benchmark set includes two groups of benchmarks: test benchmarks and toy/application benchmarks.

Test benchmarks are small programs purposely crafted to test the performance of certain architectural features. Test benchmarks include the following benchmarks:

- *Pipeline* – executes a sequence of arithmetic and register move operations, each of which uses the result of the previous instruction. This benchmark tests the impact of stalls due to data hazards. In the presence of this type of stalls, almost all of the instructions in the benchmark will be affected.
- *Pipeline-memory* – executes a sequence of load-add instruction pairs, with the ADD instruction using the result produced by the LD instruction. This benchmark tests the combined impact of memory access latency and stalls due to data hazards.
- *Loops* – runs a ten level deep loop nest, with two arithmetic operations inside the innermost loop. This benchmark tests the impact of the stalls due to control hazards, but it is also sensitive to the stalls due to data hazards, because of the dependences between the instructions in the innermost loop.

- *Memory* – executes a sequence of arithmetic operations on the elements of an array stored in the memory. This benchmark tests the impact of the memory access latency. It is also sensitive to the stalls due to data hazards.

The second group of benchmarks includes the following toy and application benchmarks:

- *Fibo* – computes the eighth Fibonacci number using recursive procedure calls.
- *Multiply* – multiplies 6x6 integer matrices.
- *Qsort* – sorts an array of 100 integers using the quick sort algorithm.
- *Stringsearch* – performs a case-insensitive search for a series of tokens in a number of strings.
- *CRC32* – computes a 32-bit cyclic redundancy check on a block of data.
- *GOL* – John Conway’s Game of Life.

Benchmarks *CRC32*, *Qsort*, and *Stringsearch* are modified versions of the benchmarks from MiBench [13], which is a freely available benchmark suite for embedded system processors.

We tested the correctness of the simulator by compiling the versions of the benchmarks that print out the results of their execution to the serial port of the Nios based system. The simulator redirects any output from the serial port to a trace file, which can be analyzed later. Therefore, we were able to verify that all the benchmarks executed correctly. For the purpose of the performance evaluation, we have disabled this printout in all of the benchmarks. The benchmarks are relatively small and the execution of the subroutines performing character output would comprise an excessively large portion of the total execution time for each benchmark, skewing the results. The benchmarks were compiled using the *gcc* compiler for Nios (version 2.9-nios-010801-20030227), with level 3 compiler optimizations. For the purpose of testing the architecture configurations with register file implemented in logic, which do not support the sliding register window, the compiler was configured to generate code without SAVE and RESTORE instructions, saving the registers to memory on each procedure call.

5 Experimental Results

The results of the timing analysis for each pipeline configuration are shown in Figure 2, in form of the processor cycle time normalized relative to the cycle time for the configuration with two pipeline stages and the register file implemented in memory. Three configurations have been tested only with the register file implemented in memory.

If the register file is implemented in memory, the only significant reduction in the cycle time is achieved by the introduction of four pipeline stages instead of three. Obviously, the increases in the number of the pipeline stages

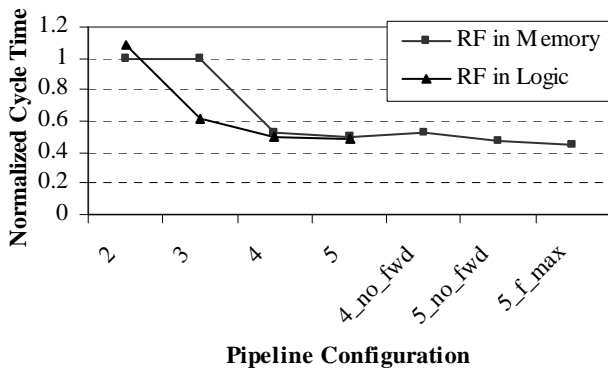


Figure 2 Normalized cycle time

from two to three and from four to five eliminate the critical path, but other near-critical paths are not removed. Pipeline configurations without the data forwarding logic and the best F_{\max} configuration achieve only a negligible reduction in the cycle time, which for most programs cannot compensate for the increase in the cycle count. Since these modifications fail in their primary purpose, we have not considered them for the configurations using the alternative register file organization. If the register file is implemented in logic, significant reductions in cycle time are achieved for increase from two to three and for increase from three to four pipeline stages.

Figures 3–6 show the cycle counts for the execution of the benchmarks. In all diagrams, the cycle count for each benchmark is normalized relative to the cycle count for the execution of the same benchmark for the configuration with two pipeline stages.

Figures 7–10 show the wall-clock execution times of the benchmarks. The execution times are calculated by multiplying the cycle counts with the cycle times determined by the timing model. In the diagrams, the execution time for each benchmark is normalized relative to the execution time of the same benchmark for the configuration with two pipeline stages.

Figure 3 shows the cycle count for the execution of the test benchmarks on the pipeline configurations implementing the register file in memory. The cycle counts for the execution of the test benchmarks directly reflect the influence of various types of pipeline stalls. Benchmark *pipeline* shows very regular behaviour, since its performance depends almost solely on the penalty for the data hazards. Benchmark *loops* is sensitive to both control and data hazards. Benchmark *memory* is sensitive to the memory access stalls, as well as data hazards. Benchmark *pipeline-memory* is also sensitive to the memory access and data hazard stalls and follows a similar pattern. However, *memory* is somewhat more sensitive to the memory stalls than to the data hazards, while *pipeline-memory* is equally sensitive to both kinds of stalls, which

explains the different behaviour of these benchmarks for the last configuration. This configuration trades an extra memory access cycle for the increased F_{\max} .

Figure 4 shows the cycle count for the execution of the toy and application benchmarks on the pipeline configurations that implement the register file in memory. Different benchmarks exhibit varying sensitivity to the stalls introduced by the more aggressive pipeline implementations. By comparing the cycle counts for different pipeline configurations, we can conclude that all benchmarks are more sensitive to the data hazard stalls than to the memory access stalls. Sensitivity to the other types of pipeline stalls varies among benchmarks. The sensitivity of the benchmarks to the two-cycle data hazards present in the 5-stage configuration without data forwarding follows an interesting pattern. These hazards may introduce stalls in the situations where an instruction reads the register written by the instruction issued two cycles earlier, which do not exist when the penalty for the data hazards is only one cycle. For some benchmarks, the number of stall cycles for this configuration will be approximately doubled compared to the 5-stage configuration with the data forwarding. However, for benchmarks where this pattern of instructions occurs frequently, the number of stall cycles may be increased much more. For example, the number of stall cycles due to the data hazards for *fibo* in the 5-stage pipeline is increased more than four times when the data forwarding is eliminated.

Figure 5 shows the execution time of the test benchmarks on the pipeline configurations implementing the register file in memory. The execution times of the test benchmarks follow the same patterns as the corresponding cycle counts, except for the step from the 3-stage to the 4-stage pipeline, where the increase in the F_{\max} overshadows all other effects. For all the test benchmarks, the 4-stage pipeline with the data forwarding is the fastest, as the increase in frequency for more aggressive pipeline configurations does not compensate for the increased number of stalls. This is not surprising, because the test benchmarks are deliberately crafted to incur large numbers of stalls.

Figure 6 shows the execution time of the toy and application benchmarks on the pipeline configurations implementing the register file in memory. For *fibo* and *multiply*, the 5-stage pipeline with the best F_{\max} is the fastest. For the other four benchmarks, the 4-stage pipeline with the data forwarding is the fastest. Benchmarks for which the 5-stage pipeline is optimal are those characterized by low sensitivity to the data hazards. From Figure 4, we can observe that among all benchmarks, *multiply* is the least sensitive to the hazards in general, while *fibo* is not sensitive to the one-cycle data hazard stalls. We conclude that the 4-stage pipeline with the data forwarding is the optimal configuration if the register file is implemented in memory.

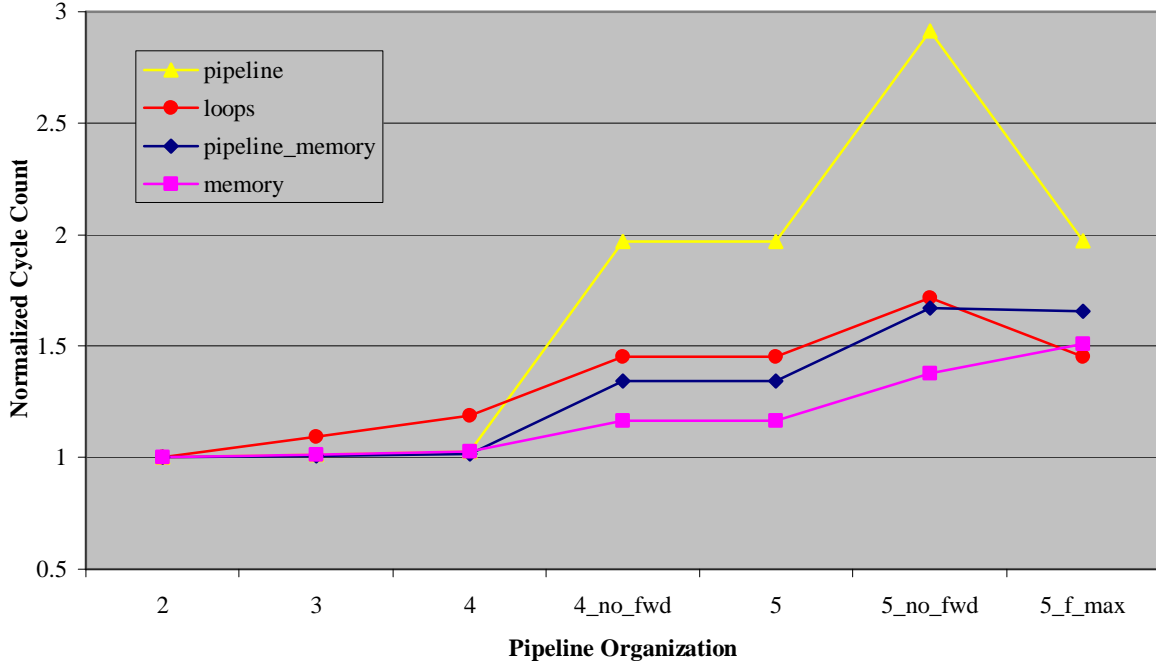


Figure 3 Cycle count for the test benchmarks on the first set of the pipeline organizations

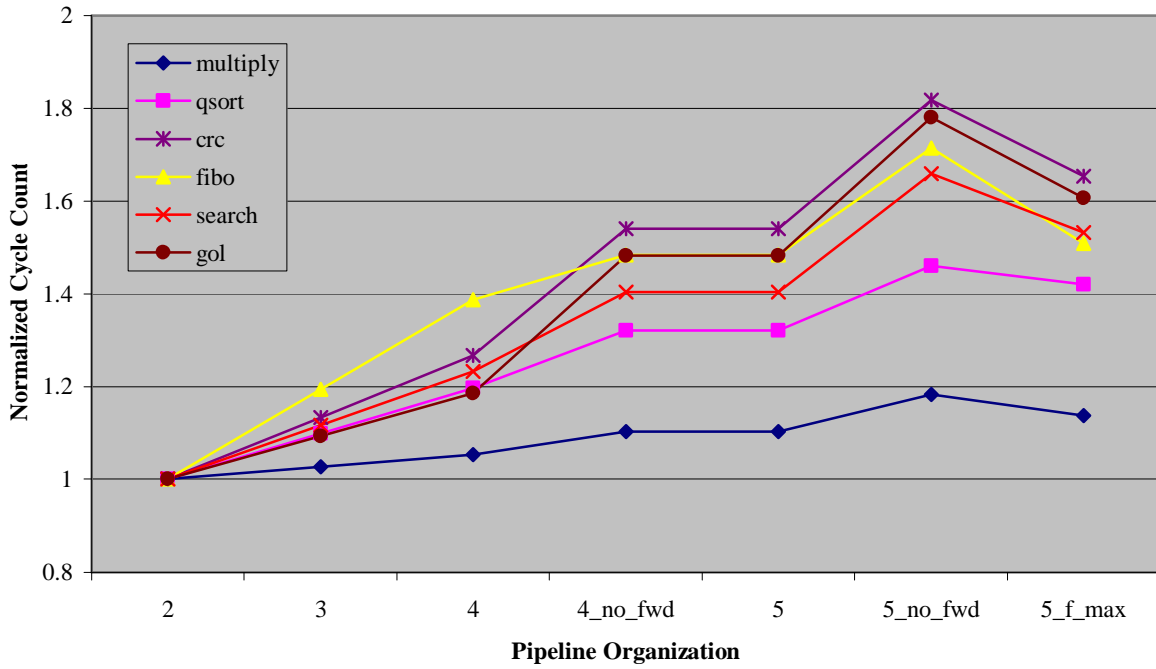


Figure 4 Cycle count for the toy and application benchmarks on the first set of the pipeline organizations

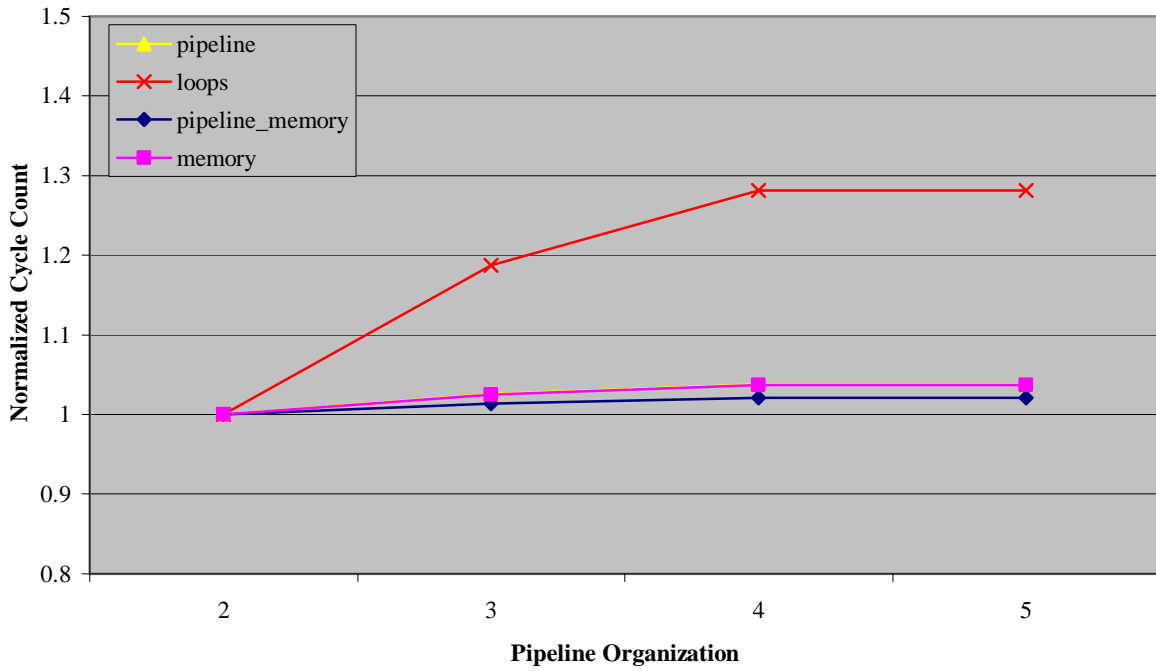


Figure 5 Cycle count for the test benchmarks on the second set of the pipeline organizations

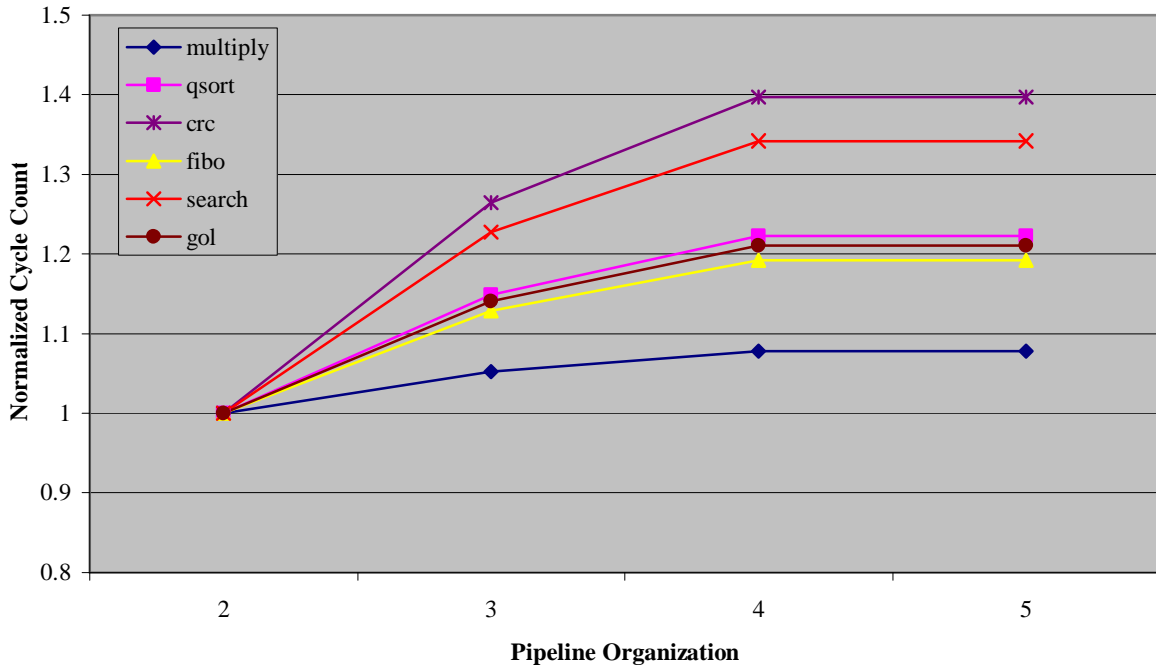


Figure 6 Cycle count for the toy and application benchmarks on the second first set of the pipeline organizations

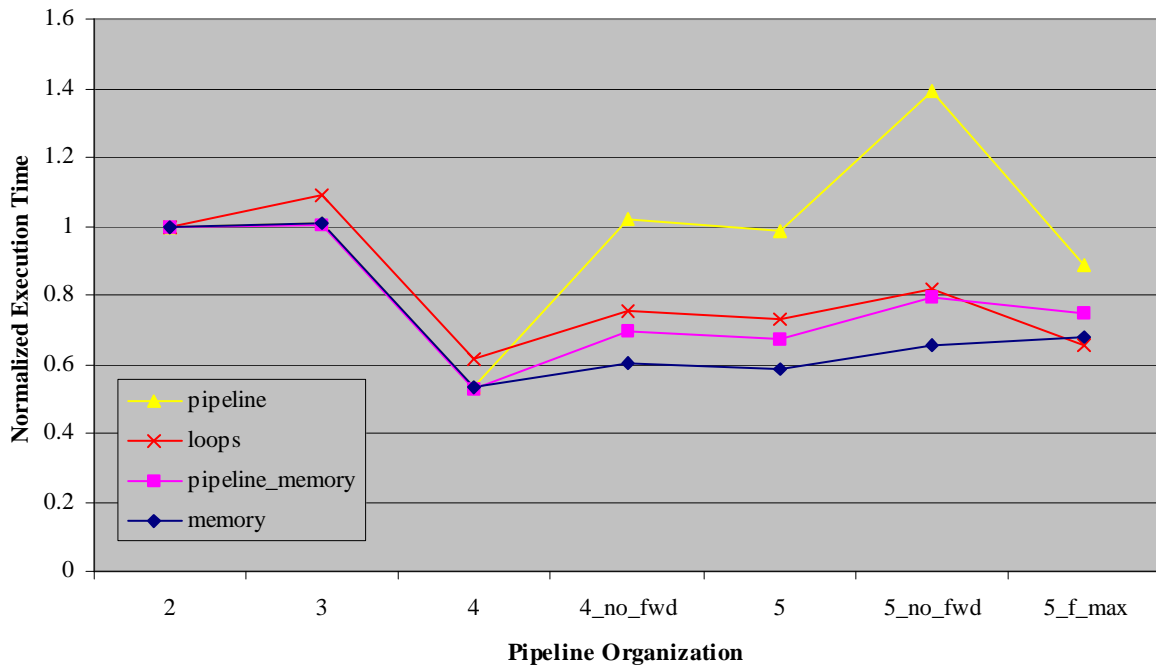


Figure 7 Execution time for the test benchmarks on the first set of the pipeline organizations

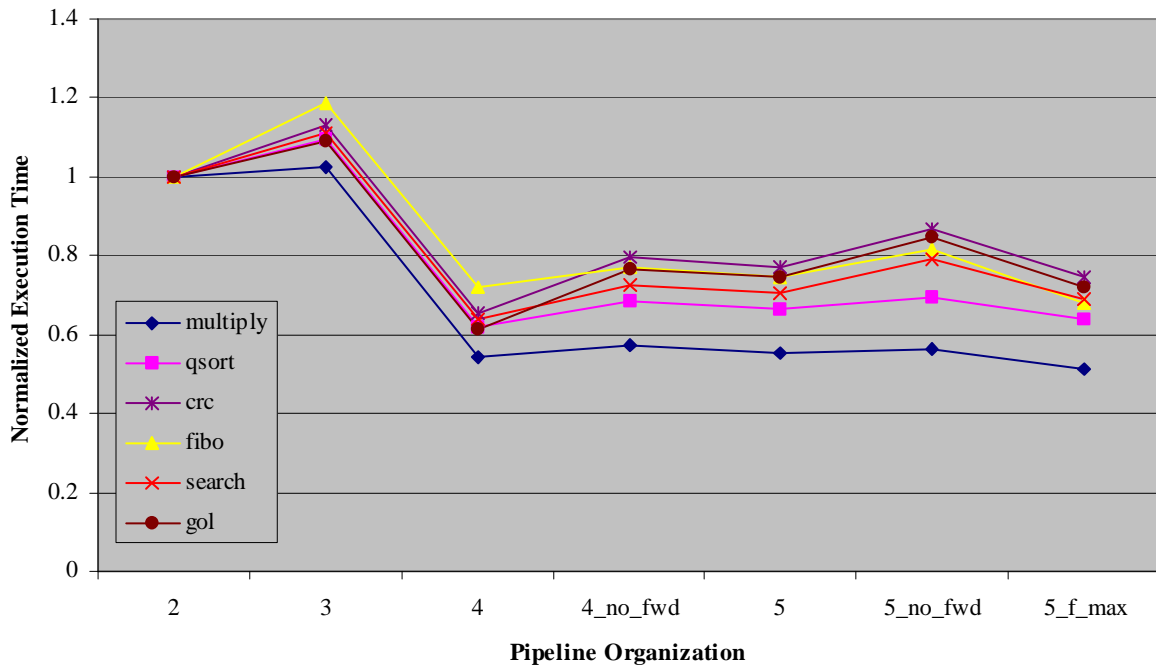


Figure 8 Execution time for the toy and application benchmarks on the first set of the pipeline organizations

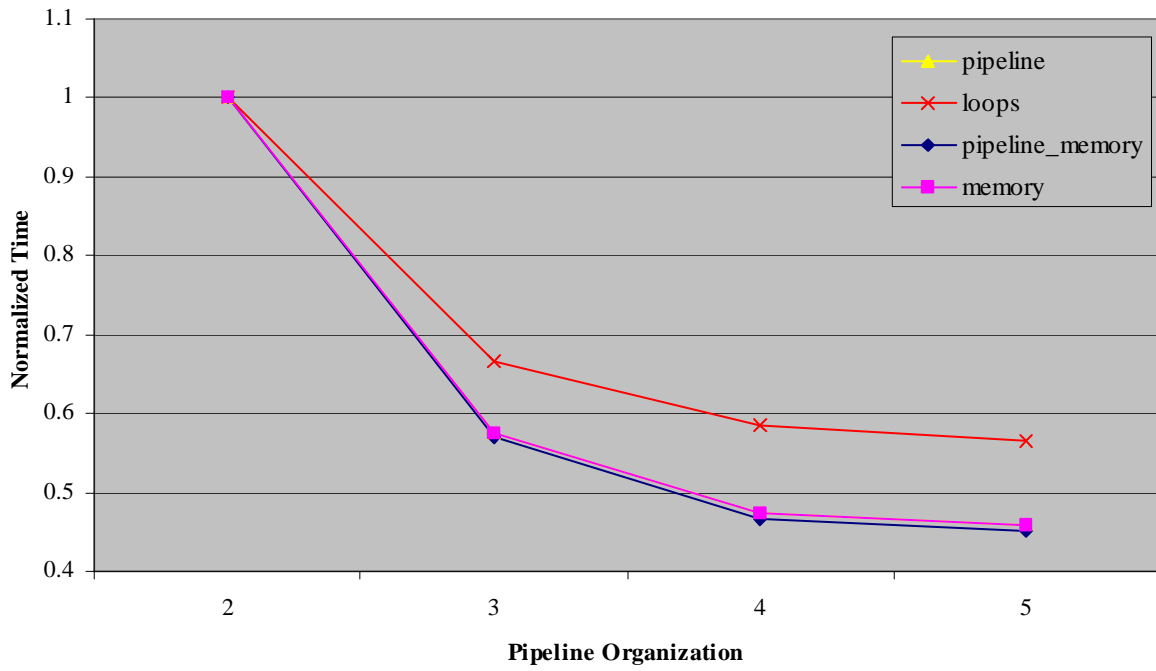


Figure 9 Execution time for the test benchmarks on the second set of the pipeline organizations

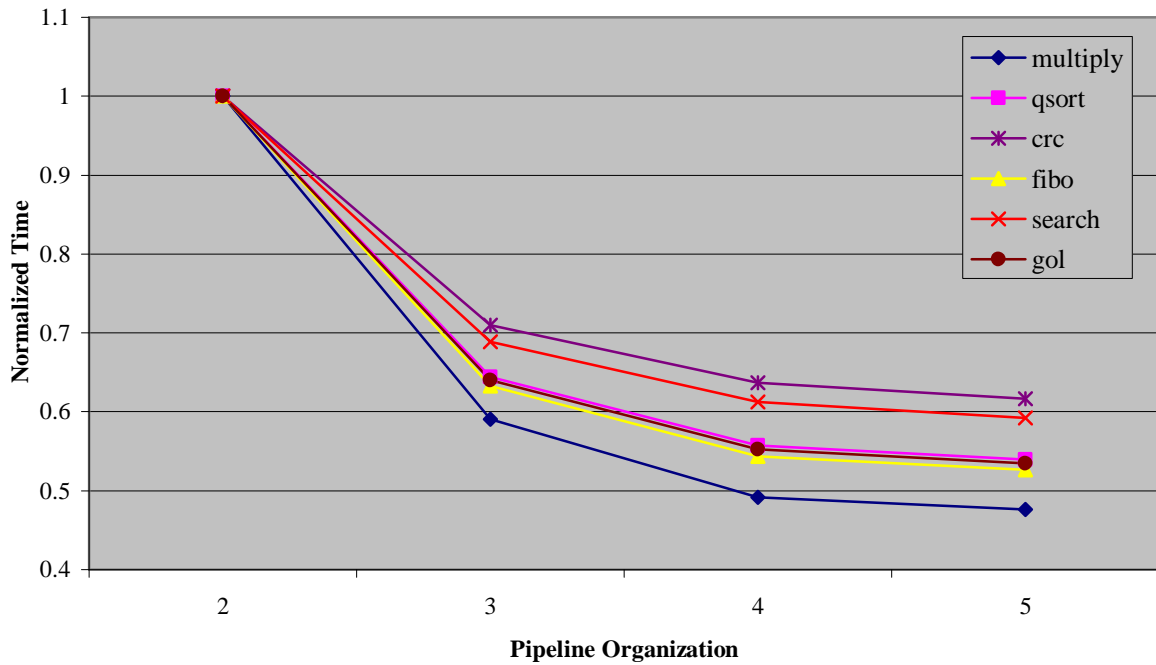


Figure 10 Execution time for the toy and application benchmarks on the second first set of the pipeline organizations

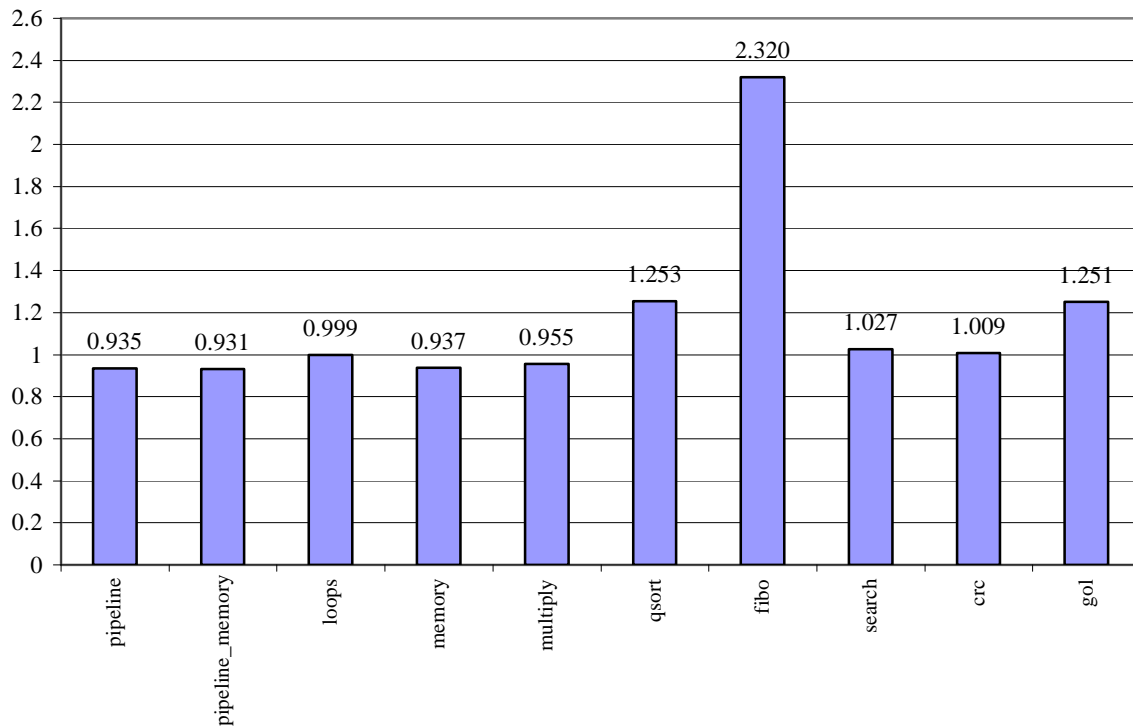


Figure 11 Comparison of the best configurations for the first and the second set of the pipeline organizations

Figure 7 shows the cycle count for the execution of the test benchmarks on the pipeline configurations implementing the register file in logic. Each of the four tested configurations does not have any stalls due to data hazards and has one stall cycle for the memory access. The only difference is the number of stall cycles due to the control hazards. The only test benchmark sensitive to the control hazards is *loop*, which explains why it is the only test benchmark whose cycle count varies significantly between these models.

Figure 8 shows the cycle count for the execution of the toy and application benchmarks on the pipeline configurations implementing the register file in logic. Again, the variation in the cycle count for each benchmark directly depends on its sensitivity to the control hazards. The sensitivity of the benchmarks to various hazards is slightly different from their sensitivity shown in the previous measurements, because they are recompiled without the support for the sliding register window.

Figures 9 and 10 show the execution time for the benchmark set on the pipeline configurations implementing the register file in logic. As the number of the pipeline stages increases, the increase in the F_{\max} is accompanied by the increase in the number of stalls due to control hazards. However, the improvements in the F_{\max} compensate for the increased number of control hazard

stalls for every benchmark, and the pipeline configuration with five stages turns out to be optimal.

Figure 11 shows the comparison between the performance of the 4-stage pipeline implementing the register file in memory and the 5-stage pipeline implementing the register file in logic. These pipeline organizations are optimal for the first and the second set of pipeline configurations. The comparison is given in the form of the execution time ratio between the optimal configurations in the second and the first set (5-stage/4-stage). For the majority of the benchmarks, the execution time for both configurations is close. Significant differences exist for three benchmarks: *fibo*, *qsort*, and *gol*. Since *fibo* and *qsort* are based on recursive procedure calls, the obvious explanation for the difference in their performance is the lack of the sliding register window in the configurations that feature the register file implemented in logic. *Gol* benchmark does not perform recursive procedure calls, but the analysis of the assembly code of this benchmark shows that it includes many procedure calls that save several registers to the stack. Therefore, the difference in the performance of *gol* can also be explained by the lack of the sliding register window.

Averaged over the entire set of the benchmarks, the performance of the configuration implementing the register file in logic is slower by 16.2%. Averaged over the toy and

application benchmarks, it performs 30.2% slower. Therefore, we conclude that the optimal configuration of the UT Nios pipeline is the 4-stage pipeline with data forwarding, with the register file implemented in memory. Also, the UT Nios that implements the register file in logic uses significantly more logic than the implementation with the register file in memory.

6 Related Work

There have been several research works aimed at development of a general-purpose analytical model for estimating the optimal number of pipeline stages for a given processor architecture. In the course of this project, we have attempted to use the analytical models presented in [14] and [15] and compare their predictions with the results of the simulation study. However, these models have turned out to be inadequate for a number of reasons. Most notably, these models rest on the assumption that the processor datapath can be uniformly divided into an arbitrary number of stages separated by registers or latches. However, the timing analysis of the UT Nios architecture clearly shows that this is not the case. Even for a small number of pipeline stages, further partitioning of the processor logic results in diminishing returns. Furthermore, the models in [14] and [15] completely ignore the effects of the increased complexity of the control unit and the overheads associated with the data forwarding logic that follow the increase in the pipeline depth. Therefore, we were unable to produce any meaningful results using these models.

The performance evaluation of the UT Nios architecture presented in [3] offers an opportunity for validating certain results of our project. Section 5.2.2 of [3] presents the performance measurements of benchmarks *fibonacci*, *qsort*, and *gol* when these benchmarks are compiled to use register windows or to use only 32 registers. Since this is also the case for the two sets of the pipeline configurations whose performance was investigated in this work, we can directly compare these results. The comparison of the results shows that the performance ratios are close, which validates our methodology.

7 Conclusions and Future Work

We have conducted an extensive simulation study aimed at evaluating the performance of various pipeline configurations for the Nios architecture. The goal of our approach was to determine the optimal configuration without having to fully implement each of the configurations. To achieve this, we have combined the results of the timing analysis, which gave an estimate of the F_{\max} for each considered pipeline configuration, and the

results of the cycle-accurate simulation on a set of benchmark programs.

Our results indicate that the optimal pipeline configuration is the 4-stage pipeline implementing the large windowed register file in memory, with full data forwarding. Comparing our results with the previous work shows that our methodology produces results consistent with the behaviour of the real systems.

An important direction for the future work would be to investigate the effects of the compiler scheduling on the performance of various pipeline organizations, particularly those whose performance when running code scheduled by the existing compiler is characterized by excessive rates of data hazards. It is unclear whether a compiler system employing an instruction scheduling algorithm specifically aimed at eliminating particular stall conditions could shift the optimal pipeline configuration towards a more aggressive, high frequency implementation.

References

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd. ed., Morgan Kaufmann Publishers: San Francisco, CA, 2003.
- [2] Altera Corporation, "Nios Embedded Processor System Development," [Online Document], Available HTTP: <http://www.altera.com/products/ip/processors/nios/nio-index.html>
- [3] F. Plavec, *Soft-Core Processor Design*, Master's Thesis, University of Toronto, 2004.
- [4] Altera Corporation, "Nios Embedded Processor, 32-Bit Programmer's Reference Manual" [Online Document], 2003 January, Available HTTP: http://www.altera.com/literature/manual/mnl_nios_programmers32.pdf
- [5] Altera Corporation, "Nios Embedded Processor, 16-Bit Programmer's Reference Manual" [Online Document], 2003 January, Available HTTP: http://www.altera.com/literature/manual/mnl_nios_programmers16.pdf
- [6] Altera Corporation, "AN 188: Custom Instructions for the Nios Embedded Processor," [Online Document], 2002 September, Available HTTP: <http://www.altera.com/literature/an/an188.pdf>
- [7] Altera Corporation, "GNUPro - User's Guide for Altera Nios," [Online Document], 2000 June, Available HTTP: http://www.altera.com/literature/third-party/nios_gnu_pro.pdf
- [8] Altera Corporation, "Stratix Device Handbook," [Online Document], 2004 January, Available HTTP: http://www.altera.com/literature/hb/stx/stratix_handbook.pdf

- [9] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers: Norwell, MA, 1999.
- [10] Altera Corporation, "Quartus II Development Software Handbook v4.0," [Online Document], 2004 February, Available HTTP: http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf
- [11] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.
- [12] B. Fort, *Simulation Tool for Soft Core Processor Performance Analysis*, Undergraduate Thesis, University of Toronto, Available HTTP: http://www.ecf.toronto.edu/~fort/UTNios_Simulation_Body.pdf
- [13] M. R. Guthaus and J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," In Proc. of the 4th IEEE Workshop on Workload Characterization, Austin, TX, 2001, pp. 3-14.
- [14] P. K. Dubey and M. J. Flynn, "Optimal pipelining", *Journal of Parallel and Distributed Computing*, vol. 8, no. 1, January 1990. pp. 10-19
- [15] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor", In Proc. of the 29th annual international symposium on Computer architecture, Anchorage, AK, 2002.