FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 17, August 2004, 199-208

A Visualization Environment for Superscalar Machines

This paper is dedicated to Prof. K. Tröndle on the occasion of his retirement

Axel Böttcher

Abstract: In this paper, we introduce an environment to visualize the internal activities of superscalar processors. This seems currently to be the dominating class of processors on the market.

A programmer or a compiler can produce optimized code only with a thorough understanding of the internal structures. This usefulness of this environment is then demonstrated for two aspects of program opimization: loop unrolling in situations with cold or perfectly warmed cache and instruction ordering. We use matrix multiplication as representative example to reflect signal processing code.

Keywords: Superscalar processors, signal processing, loop-unrolling, data dependencies, instruction ordering.

1 Introduction

With the increasing complexity of modern superscalar processors, the coding of efficient programs becomes more and more difficult. The ability of processors to simultaneously fetch several instructions and dispatch them to many parallel execution units, as well as elaborated branch prediction schemes and multilevel caches have an almost incomprehensible impact on the performance. Although the execution is nevertheless deterministic. For example, Intel has long ago stopped the publication of detailed instruction timings for their processors. Efficient compilers can only be written when the exact behavior of the target hardware is known.

The use of simulation has proven useful in performance evaluation. However, the complexity of superscalar machines demands for visualization of the internal

Manuscript received December 10, 2003.
The author is with Munich University of Applied Sciences, Department of Computer Science/Mathematics, Lothstraße 34, D-80335 Munich, Germany (ab@cs.fhm.edu).

structures to better understand the processor's behavior, see e.g. [1, 2]. In this paper we present a visualization environment for the pipeline simulator of Donald Knuth's `MMIX`-processor. This virtual processor has mainly been designed for educational purposes. However, the existing pipeline-simulator – which simulates a machine on a clock cycle basis – can be freely configured an thus be used to simulate realistic models of existing machines. This is also an ideal platform to derive perfectly reproducable results and to learn a lot about state-of-the-art hardware [3].

We tried to deduce rules for program optimization from simulation of a few situations considered as typical and representative especially for many applications in the area of signal processing.

First we will present this visualization environment. Then we will use the tool to investigate the behavior of simple iterative loops. A goal is to determine rules for program optimization that can be used in compiler design as well as for hand coding.

2 The Visualization Environment

During execution of a program, the pipeline simulator [4] can display a large amount of information as text output. Depending on the information requested, this can amount to many kilobytes per clock cycle. So there is a need for a post-processor to get the most use out of that information. We used the open source Eclipse platform [5] to implement a visualization environment. Eclipse is a lean development platform written in Java. It can be extended by own contributions in an extremely flexible manner.

At the moment the pipeline visualization consists of two main views: First, an *overview* of the configuration – see Figure 1 – showing in detail from left to right:

- The fetch buffer content, i.e. those instructions that have already been loaded but are not yet being executed.

- The execution units with all instructions being currently executed. Instructions are taken from the fetch buffer in strict program order and scheduled to the units. However, the next instruction can only be scheduled when there is a unit available that is able to execute this type of instruction. Execution itself will start as soon as all the operands are available. Those units for loading/storing (LSU) and for floating point operations (FPU) are themselves pipelined [4]. Thus a new instruction can be scheduled as soon as the previous one has entered the second stage.

- The reorder buffer (ROB) containing all instructions that are being executed or have finished execution but have not yet been committed. Execution of

instructions can end out of order, because – once on a unit – some may stall due to long execution times, or unavailability of operands (e.g. during memory accesses). In Figure 1 we see three completed instructions in the ROB (those with dark grey background) and three in execution (the second to fourth). Finally instructions are committed and thus are leaving the ROB in strict program order.

- The state of important resources like number of available rename registers, write buffer entries, or the program counter.

Furthermore an *activity view* window gives an overview of the reorder buffer's content versus time. Each pixel in the diagram of Figure 2 corresponds to one clock cycle. So the figure shows about 150 cycles (corresponding to 150ns when we assume a clock speed of 1GHz!). The bar graph in the upper half just represents the number of instructions in the ROB. Coloring is as above: dark gray for finished instructions – light gray for still executing instructions. The involved execution units can not yet be determined from this view; a double-click on the required cycle shows the details in the other window. The lower part of that figure shows activity of the memory interface which is the main reason for stalls in the pipelining. We omit the details here concerning by which line can be concluded for what reason the memory interface is busy (e.g. filling of data/code/secondary cache or write back).

In detail, Figure 2 shows the following main steps:

1. Although the pipeline is stalled due to a load operation, the memory interface is busy loading new instructions. This has just been started before the load was issued.
2. The requested data are supplied from the cache. The instructions just loaded are scheduled, some are committed and again new instructions have to be fetched from main memory.
3. A few new instructions arrive but the fetch of further instructions immediately stalls the pipeline again.
4. Some new instructions have been fetched. But same as above: new instructions are being fetched just before the issue of a load operation.

3 Investigation of Unrolled Loops

3.1 Test configuration

We have configured the simulator to behave as far as possible like the PowerPC 970 processor [6]. Since there are no timing characteristics for access to main memory

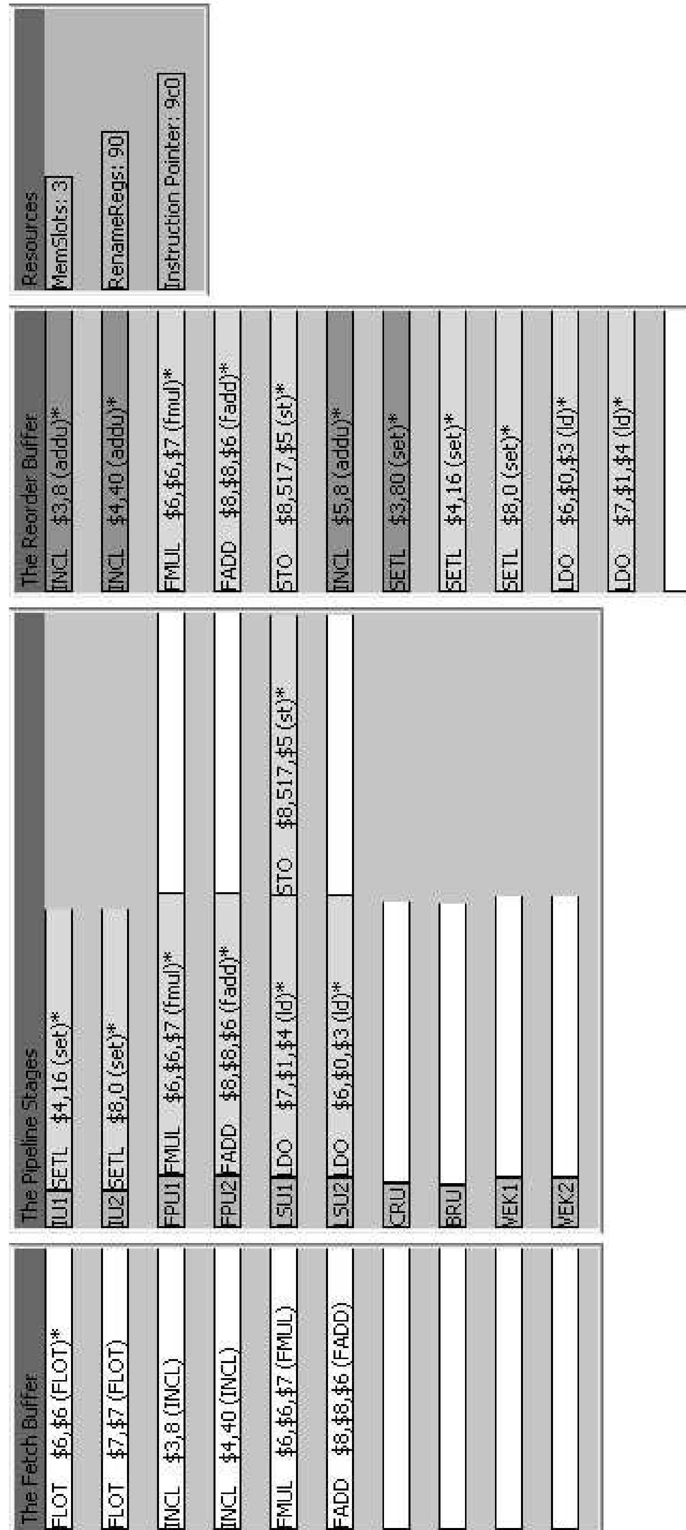

Fig. 1. Visualization of the processor configuration. Instructions will proceed from left to right.

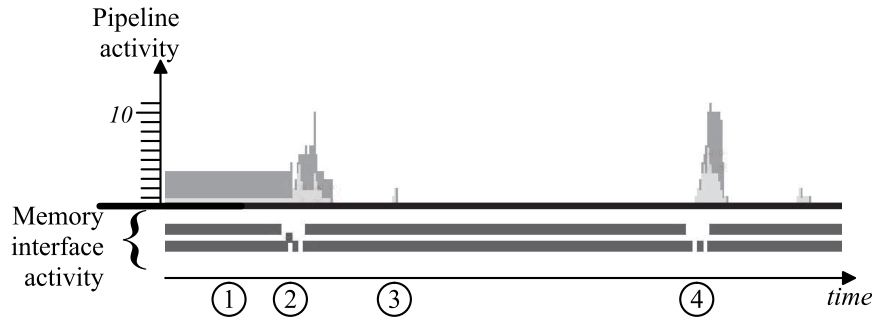

Fig. 2. Overview of the activity on the execution units and on the memory interface. The memory interface shows (top to bottom) three types of activity in this example: filling of instruction cache, filling of secondary cache, and filling of data cache.

available, we assumed three cycles to address memory and ten to read/write data.

3.2 Effects of loop unrolling

In a first step we analyzed the effects of unrolled loops which is known to be a standard compiler technique [7]. Unrolling loops results in large code but reduces the number of conditional jumps and penalty due to mispredicted branches. As example we used multiplication of two integer matrices to a floating point matrix result: $C = A \cdot B$ and implemented it in a straightforward manner:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}. \tag{1}$$

This is known not to be the optimum solution with respect to cache performance [7]. However, the standard implementation contains three nested loops, thus forming a good basis to study loop unrolling in three incremental steps.

The inner loop contains a sequence of instructions containing the following program fragment to compute one element $a_{ik} \cdot b_{kj}$:

```
1  LDO   x,A,AOff       Load first 8-byte integer: a_ik
2  LDO   y,B,BOff       Load b_kj
3  FLOT  x,x            convert to float
4  FLOT  y,y
5  INCL  AOff,ADelta    increase the offsets
6  INCL  BOff,BDelta
7  FMUL  x,x,y          multiply..
8  FADD  c,c,x          ...and add
```

Between the loop elements, the result c_{ij} has to be stored and the next step has to be initialized:

```
9   STO   c,C,COff
10  INCL  COff,8
11  SET   AOff,0*3*8  depends on position
12  SET   BOff,1*8    depends on position
13  SET   c,0         init for next iteration
```

Please note, that there is no Multiply-And-Add-Instruction for MMIX.

In a first step we multiplied two 5×5 matrices and ran each multiplication twice. In the first run none of the data are initially cached and in the second run all data will be found in the cache. This is called *cache-warming*. So the effects of cache entry replacement are excluded, which would be a study of its own [1]. Table 1 shows the results in terms of running time (in cycles) and code size. An unrolled loop causes an increased code size. In the first run this means that no

Table 1. Program performance for several levels of unrolled loops. Results are shown for cold and warm data cache.

	code size	runtime (cycles)	
program	(instructions)	cold D-cache	warm D-cache
no unrolling	26	2967	1137
one loop unrolled	55	3318	1126
two loops unrolled	231	5111	1028
completely unrolled	1126	15333	1012

instruction can be found in the code cache whereas without or with reduced rate of unrolling the code cache gets warmed up more or less quickly. The results clearly show that the effect of unrolling can only be used efficiently when the loop is repeated several times after the code cache has warmed.

3.3 Effects of instruction ordering

A further optimization strategy is the influence of the order in which instructions are assembled.

Once the instrucions to do the job are determined (in this example represented by the above code snippet), they can be rearranged in any order that respects all dependencies between them. Those dependencies can be modeled by a directed acyclic graph (DAG), as shown, for this example, in Figure 3. The vertices of that graph are labelled with line numbers of the program. Here we have to deal with two types of dependencies [7]:

1. Forwarding of results (e.g. the addition in line 8 depends on a result in line 7), known as *Read After Write* (RAW) dependency.
2. Preventing still required values in their registers from being overwritten, known as *Write After Read* (WAR) dependency. This case causes the de-

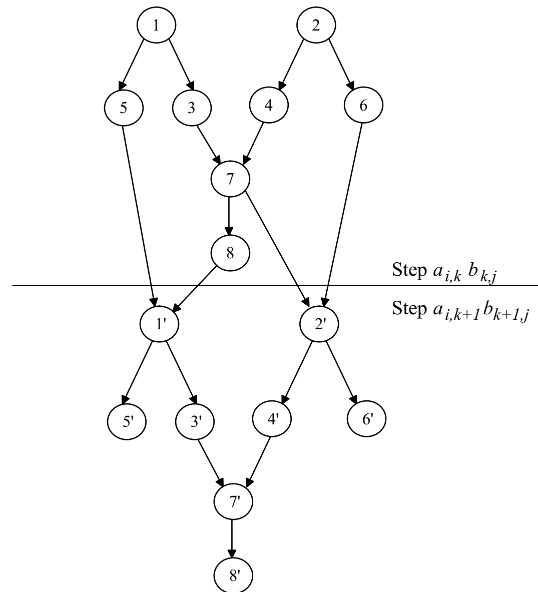

Fig. 3. The data dependency graph corresponding to the above code.

pendency between instruction in line 8 and instruction in line 1 of the next iteration (labeled $1'$ in Figure 3). Using registers more generously, generally leaves more options to order the instructions: in the example we could remove the dependency from 8 to $1'$ by replacing register x with an auxiliary register in lines 7 and 8 to forward the result. This would influence the order in which instructions will be scheduled to execution units, but not necessarily the order in which they will finally execute. This fact is due to the availability of rename registers available, that will store such intermediate values in WAR situations. Therefore such an instruction can overtake a dependent one.

The programmer (or a compiler) may arrange the instructions in any order such that the data dependencies are met. In the dependency graph, an arc from a to b means that instruction a has to be performed before instruction b because b depends on a either by RAW or WAR.

Examining the graph from Figure 3, possible orderings of the instructions are $\{1,2,3,4,5,6,7,8\}$ or $\{2,6,1,3,4,7,8,5\}$. But not $\{1,3,5,7,2,4,6,8\}$ because the dependency between 4 and 7 would be violated.

In this case, we come up with 98 possibilities to order the instructions within one iteration step to compute $a_{ik} \cdot b_{kj}$ (cf. Figure 3), excluding symmetries with respect to instructions 1, 3, 5 and 2, 4, 6. The runtimes with warmed data cache for several of these orderings are listed in Tablee 2. There are only two different

Table 2. Runtime (cycles) of one single iteration for different orderings.

instruction order	pattern	runtime (cycles)	
$\{1,3,2,4,5,7,6,8\}$	LFLFIFIF	1012	see Fig. 4a.)
$\{1,3,2,4,7,8,5,6\}$	LFLFFFII	1012	
$\{1,3,2,5,4,6,7,8\}$	LFLIFIFF	1012	
$\{1,5,3,2,6,4,7,8\}$	LIFLIFFF	1012	
$\{1,2,3,4,5,6,7,8\}$	LLFFIIFF	1108	see Fig. 4b.)
$\{1,2,3,5,4,6,7,8\}$	LLFIFIFF	1108	
$\{1,5,2,6,4,3,7,8\}$	LILIFFFF	1108	see Fig. 4c.)

results to be observed which differ by roughly 10%. This Table also lists a pattern for each ordering that gives the type of each instruction: L for a load, I for an integer and F for a floating point operation. A strategy for ordering instructions that works in many cases, is to schedule long running operations (L and F) as early as possible and to mix the type of scheduled instruction to increase the chance that a unit is available. The situation in this particular example, however, is a bit more sophisticated.

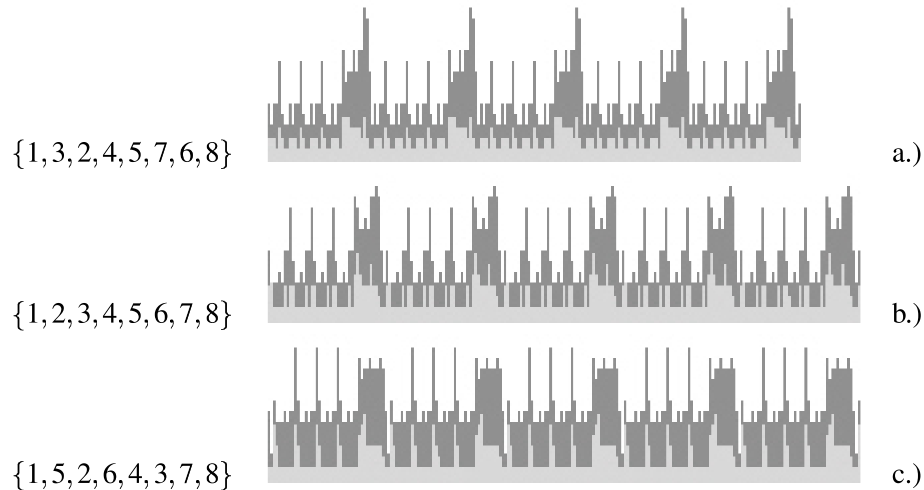

Fig. 4. Activity diagrams for three of the orderings in Table 2.

It is easy to spot in Figure 4 the instruction sequences where the processor load peaks – suggesting a closer inspection of these particular regions. Figure 5 zooms into time slices of 21 clock cycles each, highlighting details from Figure 4. Using the complete display of the processor (cf. Figure 1), careful investigations show that the difference in runtime stems from the Store Operation (line 9) at the end

of each iteration of the inner loop. The store stalls on LSU1 in its second pipeline stage for 10 cycles. When the next iteration's load instructions are scheduled simultaneously – as in case b.) at time t_1 – or with a distance of one clock cycle – as in c.) – one of them will be scheduled to LSU1 and will delay until the store completes. The latter situation can be clearly seen in Figure 1. In case a.), the loads are scheduled with two cycles in between at $t_1 \neq t_2$ and will go both to LSU2 — thus overtaking the store in LSU1.

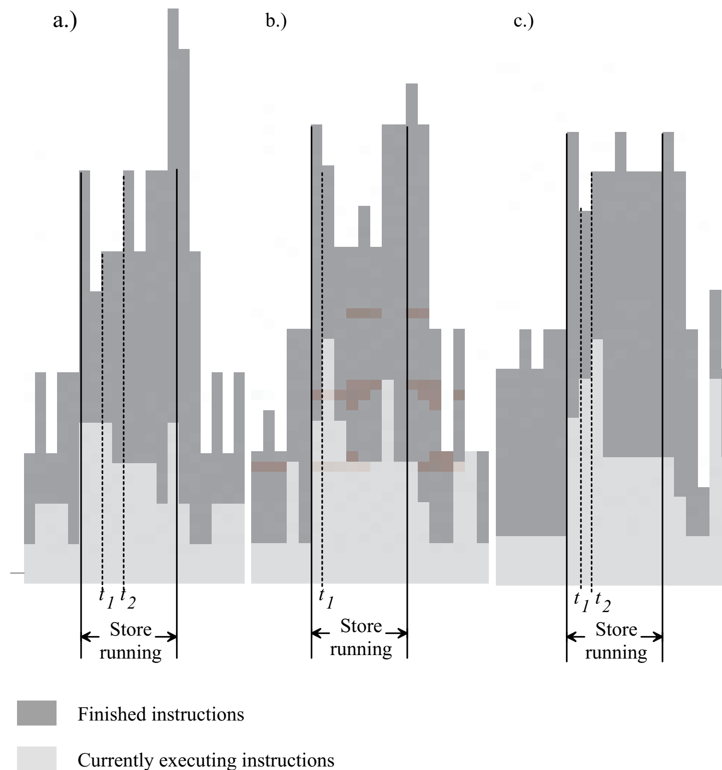

Fig. 5. Details from Figure 4a.), ab.), and c.) highlighting 21 clock cycles each. Loads are scheduled at $t_1 \neq t_2$ in a.) and simultaneously at t_1 in b.).

In general, it is inevitable to perform simulations or measurements to study and optimize the program behavior in detail.

4 Conclusions and further work

A simple piece of program for a superscalar microprocessor has been studied with respect to cache effects and instruction ordering. Understanding the details of the

execution on a clock cycle basis is quite a tricky task and all but straightforward. It demonstrated that unrolling of nested loops has to be done carefully. The visualization environment greatly supports the analysis. Trouble spots are easier to detect giving greater focus to the investigation.

The visualization environment shall be extended to cover cache details and to display more detailed information in the activity diagrams (e.g. types of instructions that are currently being executed) as well as the state of more resources. e.g. write buffers.

From `mmix-plugin.sourceforge.net`, the plugin to visualize the `MMIX`-pipeline for the eclipse platform can be downloaded. Eclipse itself is located at `eclipse.org`.

This visualization environment is also a good basis to learn a lot about hardware. It has been successfully used in the author's lectures on computer structures.

Acknowledgements

The author wishes to thank Donald E. Knuth for the nice `MMIX`-processor and for having motivated this work, and Martin Ruckert for many helpful discussions

References

[1] E. van der Deijl, G. Kanbier, O. Teman, and E. D. Granston, "A cache visualization tool," *IEEE Computer*, vol. 30, pp. 71–78, 1997.

[2] R. P. Bosch, "Using visualization to understand the behavior of computer systems," Ph.D. dissertation, Stanford University, Berkeley, CA., 2001.

[3] H. Anlauff, A. Böttcher, and M. Ruckert, *Das* `MMIX`*-Buch – Eine praxisnahe Einführung in die Informatik*. Heidelberg: Springer Verlag, 1 ed., 2002.

[4] D. E. Knuth, *MMIXware: A RISC Computer for the Third Millennium*. Berlin, Heidelberg: Springer-Verlag, 1 ed., 1999.

[5] E. Gamma and K. Beck, *Contributing to eclipse – Principles, Patterns, and Plug-Ins*. Addison-Wesley, 1 ed., 2004.

[6] P. Sandon, "Powerpc 970: First in a new family of 64-bit high performance powerpc processors," IBM, Tech. Rep., 2002.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*. San Francisco, CA.: Morgan Kaufmann Publishers, 3 ed., 2003.