Implementing microinstruction folding on the BlueJ Java optimized processor

Westmijze, Mark; Gruian, Flavius

2008

Implementing microinstruction folding on the BlueJ Java Optimized Processor

Mark Westmijze
Ewi
University of Twente
The Netherlands

Flavius Gruian
Dept. of Computer Science
University of Lund
Sweden

February 12, 2008

Abstract

This paper present the work on implementing microinstruction folding on the BlueJEP. The BlueJep is a Java Embedded Processor written entirely in Bluespec SystemVerilog. The folding model is introduced and how it is implemented. The implementation was tested on a Xilinx fpga and measurements were taken through simulation.

1 Introduction

In this paper we give a overview on implementing microinstruction folding on the BlueJep, a Java embedded processor [GW07]. Where bytecode folding has already been included in the original Picojava [MO98], we can fold instructions in the BlueJep on an other level, namely the microinstruction level. Using bytecode folding on BlueJep it would only be possible to fold on simple bytecodes which consist of only one microinstruction. Where microinstruction folding would also reduce the number of instructions of more complex bytecodes which take more than one microinstruction to execute.

This paper starts with a small introduction on the BlueJep processor in section 2. Then followed by a more elaborate explanation of folding in section 3. We continue in section 4 by examining the theoretical gain of different folding depths. Section 5 will be dedicated to the actual hardware implementation. How we synthesized our design van be found in section 6. The results of the implementation will be shown in section 7, and these are discussed in section 8. At last we draw some conclusion in section 9.

2 BlueJep Architecture

In this section we briefly introduce the BlueJep processor. The processor was inspired on the Java Optimized Processor or JOP in short. The processor

is written in a relatively new hardware description language called BlueSpec System Verilog, which is as the name suggest an extension to Verilog. In stead of Register-Transfer-Level descriptions the language is based around rules which describe functionality. Using this language in stead of VHDL or Verilog allowed us to explore more architecural designs [GW07].

2.1 BlueJep Pipeline

At the moment the BlueJep pipeline consist of six stages as can be seen in Figure 1. A detailed description of these stages can be found in [GW07] and a summerised description is included below.

Fetch Bytecode The *Fetch Bytecode* stage fetches bytecodes from the Bytecode cache. It translates these bytecodes to microinstruction addresses. Both the bytecodes and micro-addresses are forwarded to the next stage.

Fetch micro Instruction This stage fetches microinstructions from the microprogram. Using the micro-program counter (PC) stored in the register file it will fetch microinstructions from the micro-ROM and forwards them to the next stage. Foreach bytecode there is a corresponding micro-program, after the micro-program has been executed the following bytecode will be dequeued from the bcfifo.

Decode and Fetch Register After a microinstruction is dequeued is has to decoded into data moving instructions or an operation. Register values needed for the instruction are fetched in this stage. The instruction or operation is forwarded to the next stage.

Fetch Stack When a data moving instruction or operation does need one or two values from the stack they will be fetched in this stage. At this moment a data moving instruction will be forwarded to the write-back stage, while an operation will be forwarded to the Execute stage.

Execute In this stage the actual operation will be executed and the result forwarded to the last stage.

Write-back In the write-back stage the result of a data moving instruction or operation will be stored in either the register file or pushed on the stack.

3 Folding

The Java virtual machine has a stacknative instruction set, hence most instructions use the stack for data manipulation and intermediate storage for data movement. BlueJep is also a stack-based architecture, in order to execute most bytecodes as easy as possible. One of the main disadvantages of a stackbased architecture is that there are many extra data movements. For example, when

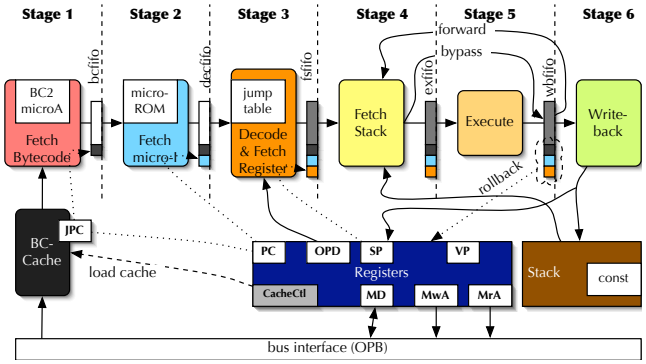

Figure 1: BlueJep Architecture

Bytecode	Length
monitorenter	9
return	9
iinc	10
ifeq	10
invokestatic	58
invokevirtual	81

Table 1: Bytecode and their microprogram size

adding two integers, the processor needs to push two integers on the stack, perform the addition on these integers and pop the top of the stack back into memory. However these four instruction can be folded into one single instruction, thus saving valuable clockcycles and memory accesses.

3.1 Bytecode folding versus microcode folding

Since the BlueJep processor runs all bytecodes on a smaller subset of stack-based operations, we have the choice to fold on either bytecode level or microinstruction level.

A simple bytecode folding algorithm is implemented in the PicoJava processor from Sun Microsystems [MO98]. From this moment on additional models for folding have been proposed, for example the POC-model in [TCaFK98], which later has been refined in [TCC00].

However folding on bytecode level would have one big disadvantage on the BlueJep processor; some of the bytecodes are so complex that they need small microprograms to emulate them. See table 1 for a few examples of how many microinstructions some bytecodes would take.

If the processor would only fold bytecodes, we can only fold on the basic

bytecodes which take one microinstruction to execute, which is only a small subset of all the bytecodes. However if we fold on the microinstruction level we would also be able to fold some of the microinstructions of the complex bytecodes. Therefore we have chosen to implement the folding mechanism on microinstruction level.

3.2 POC model used for folding

In [TCaFK98] the POC model was used to fold the bytecodes. While there are proposed enhancements to the POC model we will only use the basic POC model to fold the microinstructions in order to reduce complexity.

The POC model defines the following instructions:

Producer Produces a piece of data and pushes this onto the stack. For example: `iconst0` which pushes the constant zero onto the stack.

Operation Pops the two top entries of the stack, performs an operations on it and pushes the result back onto the stack. For example: `iadd` which pops the two top entries of the stack and pushes the addition of these two integers onto the stack.

Consumer Consumer consumes the top of the stack and can optionally store the data into the memory. For example: `istore_0` which pops the top entry of the stack and stores this in local variable 0.

Special While not an original instruction of the POC model we added the special instruction for all other instructions, for example branches and nop.

3.3 Folding patterns

In theory it is possible to construct folding patterns that are practically unlimited in length, but since we only have one ALU on the BlueJep processor we limit ourselves to the maximal length of a folding pattern with one operation. The maximal length of such a pattern is four, which is the following pattern: producer, producer, operation, consumer. See table 2 for the different folding patterns possible on the BlueJep processor.

Note that the behavior of the folded patterns themselves now looks really similar to the behavior of instructions in a RISC architecture. For example the ppoc pattern is similar to a normal RISC-operation which takes two operands as input and one operand as output.

4 Theoretical gain

The amount of foldable microinstructions depends mainly on the executed java program. While there is a small benchmark [Sch07] available for small em-

Folding Pattern	Length
ppoc	4
poc	3
ppc	3
pc	2
oc	2
po	2

Table 2: Folding Patterns on BlueJep

Patterns						Gain	
ppoc	poc	ppo	po	pc	oc	Theoretical	Total
						1.00	1.00
•						1.17	1.09
		•				1.25	1.13
•		•				1.34	1.18
		•		•		1.43	1.23
	•	•		•		1.49	1.26
•		•		•		1.54	1.29
•	•	•		•		1.61	1.33
•	•	•		•	•	1.61	1.33
•	•	•	•	•		1.64	1.34
•	•	•	•	•	•	1.64	1.34

Table 3: Folding Estimation

beddded java processors, we choose a small program which invokes a software garbage collector for simplicity.

A trace of the execution of the test program gave us the some insight in the possible gain by using folding. There are two important factors which influence the theoretical gain of folding. The first one is how many microinstructions we can eliminate and the second is how many clock cycles the processor needs on average to execute all microinstructions. It is fairly simple to make a estimation for both.

To make a simple estimation for the first factor we make the assumption that we do not need to worry about data dependencies. This means that for the theoretical gain we can fold all the patterns in table 2. Also we assume that we can always dequeue the next four microinstructions, something that might not always be the case.

For the second factor we have to estimate how many microinstructions the processor executes on average each clock cycle. This factor can be easily extracted from a simulation trace of the unmodified processor. Our test program takes *214.339* clock cycles to execute and in that time it has executed *115.829*

microinstructions, thus executing *0.54* microinstructions per clock cycle. The main reason for the low microinstruction per clock cycle is that the pipeline is stalled on every memory access. Furthermore every function call and return triggers a cache load which also stalls the pipeline.

Using both factors we can make a rough estimation of what would be the total gain of different folding pattern combinations, these can be found in table 4 and this table gives us some insight in the theoretical gain of some of different combinations of folding patterns. Clearly the most promising combination is to implement all of them. However implementing all the combinations in hardware does cost valuable area and results in a lower clock frequency. Therefore we did also look at smaller combinations of folding patterns. In the table we only list some of the more promising combinations.

5 Implementation

5.1 Overview

As noted before the BlueJep processor is written is BlueSpec System verilog, allowing us to faster explore multiple designs of the folding mechanism on the processor.

However using this language also has its disadvantages. The flexibility will most likely result in bigger and slower hardware [GW07].

5.2 Modifications

While the actual folding would only need change the decode stage there are more changes needed in order to feed the folding mechanism. For the folding the be useful, we will need a small buffer of microinstructions from which the decode stage can dequeue multiple microinstructions. Currently the fetch instruction stage delivers at most one microinstructions to the decode stage, so in most cases there will be only one microinstruction available for folding.

Therefore we made the following changes:

Multiple decode FIFOs The standard FIFOs in the BlueSpec library only support at most one dequeue and enqueue per clock cycle. Thus we need multiple FIFOs so that the decode stage will be able to dequeue more than one microinstructions per clock cycle. The instantiated FIFOs will be used as a circular buffer, allowing the fetchinstruction stage to keep enqueuing microinstructions while the decode stage dequeues microinstruction concurrently from this buffer.

One of the several advantages of BlueSpec is that normally it is fairly easy to use FIFOs. BlueSpec automatically generates the implicit stall functions needed for the correct behavior.

However our system did use multiple FIFOs in parallel. Normally a stage would only fire if there is data available in all the input FIFOs and that

all the output FIFOs are not full. The problem with this implementation was that we did not this behavior.

The most significant problem lay between the fetch instruction stage and the decode stage. The fetch instruction stage would stall if one or more of the FIFOs between these two stages were full. This is a problem in the situation were not all the the micro instructions in these FIFOs can be folded and at least one microinstruction remains. It is possible that the remaining microinstruction could be folded with following microinstructions, because these are not available the processor can only execute the remaining microinstruction.

This problem could be circumvented by using simpler FIFOs, were we had explicitly define our own stall functions.

Deeper fetch instruction stage The decode stage is now able to accept more than one microinstruction in its circular buffer. However it was not that trivial to fetch multiple microinstructions in one clock cycle. First of all it is uncertain where the next four microinstructions should come from. They could all come from consecutive microaddresses, consecutive bytecodes or a mix of the former two options. When the decode FIFOs are all empty and enough bytecodes are available in the bytecode FIFOs, our current implementation is able to enqueue four microinstructions in one clock cycle.

Multiple bytecode FIFOs The fetch instruction can only enqueue microinstructions from several bytecodes when they are available, so we implemented a circular buffer consisting of multiple FIFOs similar to the decode FIFOs.

Deeper fetchbytecode stage The new fetch byte stage can fill the bytecode buffer in one clock cycle.

All changes can be configured using macro definitions. This allowed us to compare different configurations of the folded processor with each other relatively easy.

The following options can be configured:

Fetch instruction depth Valid values are *1, 2 and 4*. This options determines how many microinstruction the fetchinstruction stage can enqueue in the decode FIFOs. Also the number of FIFOs between the fetch byte code stage and fetch instruction stage is influenced by this option. Note that the number of decode FIFOs can never be lower than this option.

Decode depth Valid values are *1, 2 and 4*. The option regulates the number of decode FIFOs. Note that this option must be at least length of the longest folding pattern.

Folding patterns This options can be any combination of the different folding patterns in table 2.

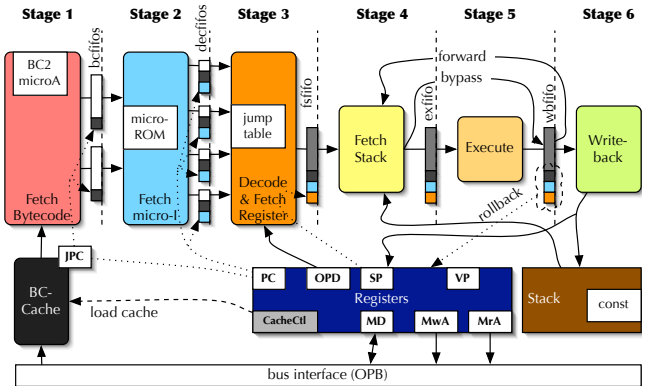

Figure 2: BlueJepFolded Architecture

See figure 5.2 for a folded configuration of the BlueJep processor with a fetch instruction depth of two and decode depth of four.

5.3 Decode stage

The decode stage is the heart of the folding mechanism, in this stage the microinstructions in the decode FIFOs are matched with the folding combinations. Using the BlueSpec language the pattern matching was achieved rather easily. See code 1 for a small section of the case statements responsible for matching the folding patterns.

5.4 Fetch instruction stage

The modifications in the fetch instruction stage were the most elaborate. As mentioned before we had to use special FIFOs so that we could generate our own stall conditions. This allowed the fetch instruction stage to also fetch new instruction when either the bytecode buffer was not completely full or the decode buffer was not completely empty.

First the fetch instruction stage determines where the next microinstruction come from. This means that the stage first checks the current microinstruction for the NXT bit. When this bit is set, the next microinstruction will come from the next bytecode, if not the following microinstruction is the next from the microcode. This step is performed as long as there is another bytecode available and a free place present in the decode buffer.

6 Synthesis and Simulation

The BlueSpec System verilog code was converted to verilog using the BlueSpec 2006.11 compiler.

```
case(combine) matches
  tagged Combine { i_0: tagged Push { src: .srcb},
                   i_1: tagged Push { src: .srca},
                   i_2: tagged Oper { op:  .op},
                   i_3: tagged Pop  { dst: .dst}}:
    action
  infifos[decselect + 0].deq();
      infifos[decselect + 1].deq();
      infifos[decselect + 2].deq();
      infifos[decselect + 3].deq();

      Context crtctxt = Context { jpc: mi_2.jpc,
                                  mpc: mi_2.mpc,
                                  sp: sp};

      fsfifo.enq(ToFetchStack {
        ctxt: crtctxt,
        dst: getDestination(dst),
        src: tagged TwoAddr {
          ra: getSource(srca, mi_1.jpc),
          rb: getSource(srcb, mi_0.jpc),
          op: op
      }});

  endaction

  ...

endcase
```

Code 1: Decode stage

These files were synthesized using the Xilinx Ise 9.1 tools.

The target for synthesis is the Virtex 5 platform from Xilinx. Where applicable synthesize options were optimized for speed.

All the simulation were carried out using the simulation option from the BlueSpec compiler. Using chipscope a trace from all opb transfers for the original processor was made to ensure that the simulation traces were cycle accurate.

7 Results

The results of some our test can be found in table 7. We choose only to show the results of the more promising tests for total of folding patterns and their subsets.

8 Discussion

As seen in table 7 none of the configurations perform better than the unmodified version of the BlueJep processor. The main reason is the enormous drop in clock frequency due to the added hardware required for the folding mechanism.

For example, let us compare some configurations to each other. Let us begin with the configuration with fetch instruction depth two, two decode fifo's and no folding patterns with the same configuration but now implementing all the folding patterns. In this case the clock frequency drops by 31.8 percent.

In the same way we can compare the original BlueJep processor to the configuration that has fetch instruction depth four and has four decode fifo's and no folding patterns. In this case the clock frequency drop by 40.7 percent.

Furthermore, it does get even worse when we implement all folding patterns, a fetchinstruction depth of four and four decode fifo. In that case the clock frequency drops by 47.0 percent in comparison to the original BlueJep processor. While the clock cycle count has only dropped by 19.2 percent, resulting in a total loss of 33.0 percent.

However both stages responsible for this loss were written pretty straightforward. We believe it should be possible to further pipeline these stages to regain some or all of the clock frequency. For example, the fetch instruction stage could be split in two by letting one stage fetch the microinstruction from the microrom and let the second stage decide which of them should be enqueued into the decode fifo's.

In the same way the decode stage could be split in half by letting one stage check for folding and let the other stage do the actual decode of the microinstructions.

9 Conclusion

Since none of our test configurations showed any promise of actually being faster than our original BlueJep processor we conclude that folding on microinstruction

Table 4: Folding results

		Patterns									Gain[a]			
Fetchinstr. depth	Decode fifo's	ppoc	poc	ppo	po	pc	oc	Clock cycles	Clock frequency (Mhz)	Size (ratio)	Clock cycles	Clock frequency	Size	Total
1	1							211843	225.3	11	1.00	1.00	1.00	1.00
2	2							214438	205.7	14	0.98	0.91	0.78	0.89
2	2				•	•		198759	189.8	15	1.06	0.84	0.73	0.89
2	2				•	•	•	195559	181.6	15	1.08	0.80	0.73	0.86
2	4							214391	192.4	15	0.98	0.85	0.73	0.83
2	4				•	•		189739	164.0	17	1.11	0.72	0.64	0.79
2	4				•	•	•	189253	160.2	17	1.11	0.71	0.64	0.78
2	4			•	•	•		182787	155.6	17	1.15	0.69	0.64	0.79
2	4			•	•	•	•	182301	143.5	17	1.16	0.63	0.64	0.73
2	4		•					209305	176.3	17	1.01	0.78	0.64	0.78
2	4		•	•	•	•	•	180278	133.8	19	1.17	0.59	0.57	0.69
2	4	•						214299	157.6	18	0.98	0.69	0.61	0.67
2	4	•			•	•		190637	144.5	19	1.11	0.64	0.57	0.71
2	4	•	•	•	•	•	•	180264	131.2	19	1.17	0.58	0.57	0.67
4	4							214475	133.5	21	0.98	0.59	0.52	0.57
4	4				•			198887	136.0	22	1.06	0.60	0.50	0.63
4	4				•	•		192149	130.7	23	1.10	0.58	0.47	0.63
4	4				•	•	•	191480	130.2	23	1.10	0.57	0.47	0.62
4	4			•	•			189946	135.0	24	1.11	0.59	0.45	0.65
4	4			•	•	•	•	182441	122.8	24	1.16	0.54	0.45	0.62
4	4		•		•	•	•	183288	129.0	24	1.15	0.57	0.45	0.65
4	4		•	•	•	•		177020	125.0	25	1.19	0.55	0.44	0.65
4	4		•	•	•	•	•	176351	115.1	25	1.20	0.51	0.44	0.61
4	4	•						210701	133.8	23	1.00	0.59	0.47	0.59
4	4	•	•	•	•	•		171791	118.7	26	1.23	0.52	0.42	0.63
4	4	•	•	•	•	•	•	171122	119.5	26	1.23	0.53	0.42	0.65

[a]Relative to unmodified version of the BlueJep processor, which is shown in the first column.

level is not a viable option to increase the speed of the processor.

Furthermore we suspect that even with further pipelining the processor might only become marginally faster than the unmodified version. So even in that case the small increase in speed would in most cases not outweigh the additional hardware, because without further pipeline the processor is already 136 percent bigger than the unmodified version. For some applications using two BlueJep processors in parallel might be a more viable option.

References

[GW07] F. Gruian and M. Westmijze. Bluejep: A flexible and high-performance java embedded processor. *JTRES*, 2007.

[MO98] Harlan McGhan and Mike O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, 1998.

[Sch07] Martin Schoeberl. Jop bencmark. `http://www.jopdesign.com/perf.jsp`, 2007.

[TCaFK98] Lee-Ren Ton, Lung-Chung Chang, and Chung-Ping Chung andMin Fu Kao. Stack operations folding in java processors. 1998.

[TCC00] Lee-Ren Ton, Lung-Chung Chang, and Chung-Ping Chung. Exploiting java bytecode parallelism by enhanced poc folding model (research note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 994–997, London, UK, 2000. Springer-Verlag.