Low Power Microarchitecture with Instruction Reuse

Frederico Pratas
INESC-ID/IST
Rua Alves Redol, 9, 1000-029
Lisboa, Portugal
fcpp@inesc-id.pt

Georgi Gaydadjiev
CE Lab/TU Delft
Mekelweg 4, 2628 CD
Delft, The Netherlands
georgi@ce.et.tudelft.nl

Mladen Berekovic
TU Braunschweig
Braunschweig, Germany
berekovic@ida.ing.tu-bs.de

Leonel Sousa
INESC-ID/IST
Rua Alves Redol, 9, 1000-029
Lisboa, Portugal
las@inesc-id.pt

Stefanos Kaxiras
University of Patras
Patras, Greece
kaxiras@ee.upatras.gr

ABSTRACT

Power consumption has become a very important metric and challenging research topic in the design of microprocessors in the recent years. The goal of this work is to improve power efficiency of superscalar processors through instruction reuse at the execution stage. This paper proposes a new method for reusing instructions when they compose small loops: the loop's instructions are first buffered in the Reorder Buffer and reused afterwards without the need for dynamically unrolling the loop, as commonly implemented by the traditional instruction reusing techniques. The proposed method is implemented with the introduction of two new auxiliary hardware structures in a typical superscalar microarchitecture: a *Finite State Machine* (FSM), used to detect the reusable loops; and a *Log* used to store the renaming data for each instruction when the loop is "unrolled". In order to evaluate the proposed method we modified the *sim-outorder* tool from *Simplescalar v3.0* for the PISA, and *Wattch v1.02* Power Performance simulators. Several different configurations and benchmarks have been used during the simulations. The obtained results show that by implementing this new method in a superscalar microarchitecture, the power efficiency can be improved without significantly affecting neither the performance nor the cost.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*

General Terms

Design, Performance

CF'08, May 5–7, 2008, Ischia, Italy.

Keywords

Power Reduction, Superscalar Processor, Loop Reusing Technique, Reorder Buffer Optimization.

1. INTRODUCTION

Three decades of history of microprocessors report truly remarkable technological advances in the computer industry. This evolution closely followed the well-known Gordon Moore Law [8]. Aiming ever-faster microprocessors, several strategies that focus in exploiting *Instruction-Level Paralelism* (ILP) have been adopted, namely: deeper pipelining, multiple execution units, wider fetching mechanisms and speculative execution have been used in the microprocessors design [4]. Today's superscalar processor microarchitectures prove the effort that has been done to exploit ILP. Due to this evolution, power consumption has also become one of the most important parameters in the design of such complex systems [13].

Nowadays, *instruction reuse* (IR) techniques are used to improve the power efficiency and ILP in superscalar microarchitectures. Reuse methods have been developed at different stages: at the fetch/decode stages, for instance through trace caches [10] and dynamic instruction reuse [14]; and at the execution stage, using mechanisms such as the issue queue buffering [6], the trace reuse [18] and the execution cache [16]. All of these methods reuse instructions that have been previously fetched from the instruction cache and that are still available in the processors core, in order to prevent the fetch and/or the passage of instructions through processing stages; for instance by reusing instructions at the execution stage, the front-end of the processor can be gated-off. Therefore, such methods can improve power efficiency by exploiting repeated operations during the execution of programs, like the frequent existence of loops in real programs.

This paper proposes an enhanced microarchitecture which implements a new IR method at the execution stage to reduce the power consumption in a superscalar processor. Besides reducing the number of accesses to the memory, the proposed method also decreases the power consumption by reusing the instructions directly from the *reorder buffer* (ROB) and thus avoiding the flow of redundant data in the ROB by preventing explicit unrolling of simple dynamic loops with a single control path. To implement the

Table 1: Summary of IR methods characteristics.

Method	Characteristics					
	Register renaming	Data origin	Data structures	Look-up schemes	Explicit loop unrolling	Typical size (kbits)
Execution Cache	bypassed	EC	any trace	✓	✓	128
Trace Reuse	✓	ROB	basic blocks	✓	✓	1.5
Issue Queue	✓	IQ	loops	no	✓	2
CLU	✓	ROB	loops	no	no	1.5

proposed IR method, two new hardware structures have to be introduced: a controller, to detect the reusable instructions, and a small buffer, to store the renaming data generated in the *Register Alias Table* (RAT). Simulation results reveal that the power efficiency can be improved up to 10%.

The remainder of this paper is organized as follows. Section 2 reviews previously proposed IR techniques. Section 3 analyzes the opportunities and challenges of reusing the instructions from the ROB and introduces the proposed method. Section 4 presents the implementation details of the proposed method in the microarchitecture, including the new hardware structures that are required. Finally, the performance evaluation results are presented in section 5 and section 6 concludes this paper.

2. ANALYSIS OF KNOWN INSTRUCTION REUSE METHODS

Three main techniques have been proposed to reuse instructions in superscalar microarchitectures at the execution stage: *i)* Execution Cache; *ii)* Trace Cache; and *iii)* Issue Queue buffering.

2.1 Main characteristics of each IR method

The method proposed by *Talpes et al.* [15] consists on inserting a trace-cache-like mechanism deeply in the pipeline of the processor to reduce the instruction processing path, and thus improving the performance. Using this trace-cache, after the Issue stage - designated by *Execution Cache* (EC) - the instructions already fetched, decoded, and renamed can be stored in dynamic order in the cache structure and reused directly as a consequence of a hit. This mechanism comprehends mainly two phases: *i)* a trace is built and stored in the execution cache in issue order, while instructions are executed from the Issue stage; *ii)* a search is performed in order to find a trace that starts at the same point as the actual execution point; in the case of a hit, the instructions are executed directly from the execution cache sequentially.

The method proposed by *Yang et al.* [18] reuses the instructions directly from the *reorder buffer* (ROB). Since the required information, that is produced by the fetch and decode stages and is associated to the in-flight instructions, is stored in the ROB, we can reuse it whenever a reusable instruction is identified, by directly forwarding it from the ROB to the renaming stage. The reused instruction is then normally renamed, reinserted in the ROB, executed, and committed following a shorter instruction processing path. Since the ROB behaves like a cache, a look-up step is needed to find reusable instructions. Instructions under fetch and in-flight instructions are ordered as traces of basic blocks and each basic block is stored in the ROB in program order. Therefore, if a match is found for the starting instruction of a basic block, then all instructions until the next basic block will also be reused. A *Reuse Identification Unit* (RIU) is used in order to track the content of the ROB at the block level. The RIU records the starting *Program Counter* (PC), the corresponding ROB entry index and the size of the respective block traces.

Finally, the *Issue Queue* method [6] reuses the instructions directly from the *Issue Queue* (IQ) creating a new instruction processing path. The proposed issue queue design has a mechanism to dynamically detect and identify reusable instructions belonging only to tight loops. When a tight loop is detected, its instructions are buffered from the Issue Queue instead of being fetched from the memory. This method is composed by the following four main components: the loop detector, that checks for conditional branches and jump instructions (backward branch/jump) in a loop and if the loop fits in the Issue Queue; the buffering mechanism, that keeps the instructions in the queue even after being dispatched to the functional units; the scheduling mechanism, which controls the keeping/removing of the instructions in the queue; finally, the issue queue can change back to the non-reusing state when an ongoing buffering is revoked and/or a misprediction occurs. The implementation of this method is supported by a dedicated *Finite State Machine* (FSM) that controls the state of the IQ, and also by using additional information stored in the IQ.

Table 1 summarizes the main characteristics of the methods previously described and the one proposed herein, which will be discussed in the next section, namely: the hardware structures used for reuse in each method and the kind of data structures detected and reused.

2.2 Comparative Analysis

From the three approaches referred above, only the method proposed by *Talpes et al.* (see section 2.1), based on the Execution Cache, is able to reuse the register renaming information. Besides, when a trace is built in the *Execution Cache* the instructions lose their original logical order and can only be retrieved on a sequential basis. Therefore, with each change of trace the processor must perform a look-up step in the *Execution Cache* to either find a new trace or retrieve to an in-order execution, leading to some breaks in the potential ILP.

The *Trace Reuse* method based on the information in the ROB has some drawbacks, namely: *i)* when there is a switch to the conventional path, an additional penalty cycle is spent in searching the RIU in order to ensure that no match exists for the given predicted fetch address; *ii)* for a loop with N active iterations and unrolled in the ROB, the RIU has at least N entries occupied for each repeated basic block (due

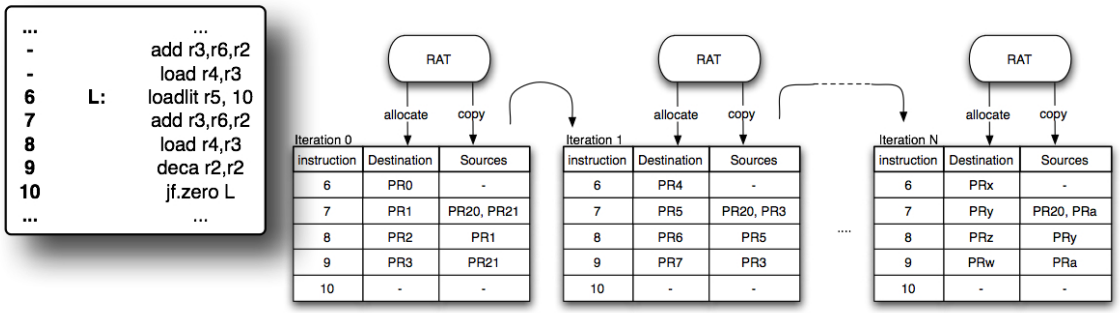

Figure 1: Loop unrolling example; PRi - Physical Register i.

to all branch instructions being inserted in the RIU entries); *iii)* the time to reuse a block may be to short, because it can only be reused until the first instruction in the block is committed. A possible solution for the last handicap is to increase the ROB size; it increases the probability for the ROB to contain multiple copies of the same block. Consequently, the probability for that block to be captured is also higher. Nevertheless by increasing the ROB size we also increase the power consumption.

Contrasting with the previous methods, the *Issue Queue* method only reuses specific types of code structures (*simple loops with a single control path*), and not generic traces. The hardware overhead is very low, so the power consumption in the additional structures is also low regarding to the obtained power reduction, for example, for regular processing. Nevertheless, breakdowns in the performance can occur during the recovery procedures when switching between states. As an example, this situation can occur whenever a program has a lot of potential loops but with a small number of iterations, because the system has to switch a lot of times between different states.

The method proposed in this paper, and also presented in Table 1, is designated as *Constrained Loop Unrolling* (CLU). Although it gathers some of the characteristics from the previously presented methods, it mainly reuses instructions from the ROB as in [18] and exploits the fact that some parts of a program consist of repeated traces of code with a single control path (*tight loops*), by performing loop detection as in [6]. Moreover, it has the particular characteristic of performing dynamic scheduling with implicit loop unrolling, as it will be explained further in the next section.

3. PROPOSED INSTRUCTION REUSE METHOD

Loops in programs provide some *a priori* information, namely the fact that a set of instructions has a high probability of being executed several times. When a program loop has high iterations count, the power consumption in the ROB significantly rises, due to redundant data being stored in the ROB during the explicit loop unrolling (high number of accesses). Furthermore, the ROB usually has to be a large structure with multiple *Input/Output* (I/O) ports.

In order to simplify the design of our proposal we focus this work on simple loops with a single control path, as the example provided in Figure 1. The repetition of the instructions that constitute each loop iteration can be exploited by reusing the instructions while they are present in the processor core. Implementing such idea requires that information about loop instructions is kept in the microarchitecture, for example by using additional hardware structures. This would be a straight approach when the reuse of instructions is carried out at the fetch stage. However, instruction reuse in a deeper level of the pipeline, such as the execution stage, would allow to gate-off the complete front-end of the processor, thus potentially saving more power. Moreover, if the reusable information is kept in small hardware structures, there is also the possibility to improve ILP by using wider and faster buses during the reuse phase.

Based on this analysis, a new method to perform IR at the execution stage of a superscalar processor is proposed: the CLU. The main idea of this method is to implement IR by providing instructions from the ROB when a small sized loop is detected. The proposed CLU method uses the information in the ROB in a different way: when processing a loop there is no need to repeat the same information for each iteration, as actually happens when the loop is unrolled in the ROB. Instead, only the information related to the renaming of the destination registers is stored. Such information is important to free the allocated physical registers at the commit stage or when a misprediction occurs, because in many cases the *physical register file* (PRF) is separated from the ROB [13]. Thus, a new structure is needed to store the information related to the register renaming, for supporting implicit unrolling. Mainly, this structure is very small and its power consumption is very low compared with the one of the ROB. The proposed structure stores register renaming information in each iteration of the loop, and an additional counter is needed for counting the executed instructions in each iteration of the loop. When an instruction of a certain iteration is finished, the counter is incremented and only when an iteration is finished the instructions can be committed since commit has to be kept in order. During the instruction reuse, static branch prediction is used to simplify the recovery mechanism.

Three different phases can be identified for the detection of instructions to reuse and to control the operation of the proposed CLU method: *i)* instructions are fetched until a reusable loop is found; *ii)* the loop is buffered and fixed in the ROB; *iii)* the instructions are directly provided by the ROB. It is worth noting that the instructions of the loop can be directly reused from the ROB, since they are stored in program order.

As in the previous methods, in the CLU method the front-

end is decoupled from the usual instruction path when the alternative path, shorter than the main one, starts providing the reusable instructions. In all the methods this is the main source of power savings, because the fetch mechanism is gated-off. Table 1 provides an overview of the main characteristics of the CLU method and compares them with the ones of the remaining methods. In the CLU, although the rename stage is not bypassed, all the unnecessary look-up schemes used in previous works are avoided. Moreover, in contrast with the IQ method described in section 2, only one iteration of the loop is buffered at a time. During the scan phase, a potential loop is only considered when a group of sequential instructions between two backward branches with the same address is detected.

The ROB is a structure that consumes a relevant amount of power [7] because, as stated before, it is a large and complex structure accessed simultaneously in several stages of the processor, i.e., it needs several I/O ports and it is used for each and avery instruction. If the loop instructions are not inserted in the ROB during the reuse phase, i.e., the loop is not unrolled inside the ROB, the number of accesses to the ROB is reduced and consequently the overall power consumption. Moreover, to implement this process we do not need to store the redundant information mentioned before, but only the renamed destination register in a small structure, as will be explained in the following section.

4. INTEGRATION OF THE PROPOSED CLU METHOD IN THE MICROARCHITECTURE

4.1 Microarchitecture

The microarchitecture has to be modified, in order to support the proposed CLU instruction reuse method. The following mechanisms have to be implemented in the superscalar microarchitecture: *i)* loop detection, to detect a loop fetched from the memory and activate the reuse method; *ii)* loop buffering, to buffer the loop in the ROB before it can be reused; *iii)* loop reuse, which should enable the *Log* structure (docked to the ROB) to start storing the information needed and to unroll the loop.

Therefore, two structures are introduced in the typical superscalar microarchitecture to support these mechanisms: the *Log* and the *Small Loop Finder* (SLF), as shown in the block-diagram presented in Figure 2. The SLF implements an FSM that controls the loop detection, the loop buffering and the loop reuse. The *Log* structure is a buffer where the instruction's renaming information is stored during loop reuse, i.e., it keeps track of the physical registers allocated in each loop iteration. In this case, the registers used in a specific iteration can only be freed and the corresponding instructions committed if the iteration was correctly executed (a misprediction did not occur) and all the previous iterations were committed.

The recovery misprediction mechanism has to be modified. In case of a misprediction the reuse mechanism has to proceed as follows:

- the registers allocated for the instructions after the mispredicted branch are freed and the respective fields in the *Log* are invalidated; when a misprediction occurs some instructions are still in execution and will arrive later at the write-back stage. A tag is used to validate this instructions;
- the valid instructions still inside the *Log* are executed and committed normally;
- the FSM controlling the loop detection does not insert a new loop in the *Log*, while instructions from a previous loop are in use to guarantee that the old loop finishes correctly;
- when a *Log* reaches the last iteration, the *Reused* fields in the ROB are reset and the FSM can introduce a new loop.

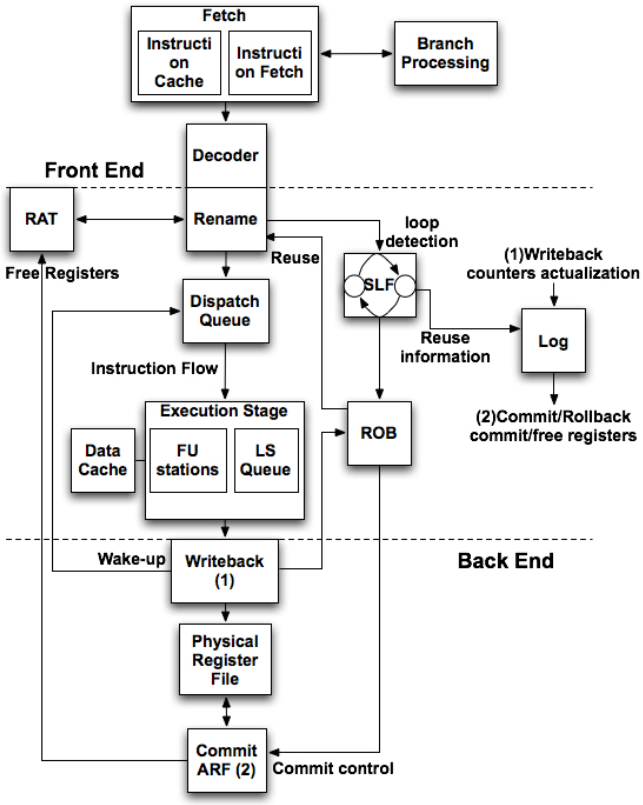

Figure 2: Microarchitecture with the CLU method.

4.2 The Finite State Machine

The identification of small loops is accomplished by a detection mechanism based in a simple FSM with four states (see Figure 3). Functionally, the four states correspond to the sequence depicted in Figure 4. In the first phase, it scans the instructions to find two sequential branch instructions in the same path with the same PC value. There are two separate steps during this phase: in the first step the FSM tries to find a branch instruction and in the second step it checks the basic block ahead to find a reusable
loop. Only the loops containing valid instructions, i.e., the ones that do not contain any jump, call, indirect branch and/or return instructions, can be reused (example in Figure 4). If these conditions are met, the possibility of a pseudo basic-block to be executed several times is high, and therefore it can be reused.

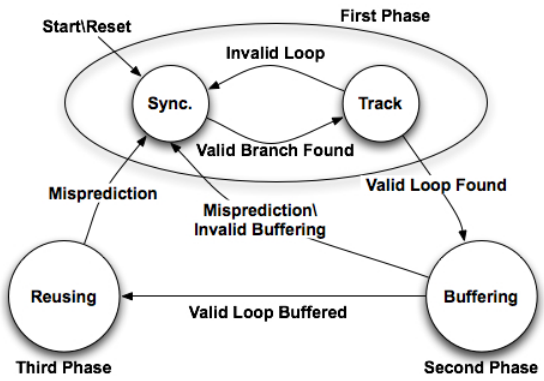

Figure 3: Loop detection Finite State Machine.

The loop has to be identified in an in-order processing phase, which means that it has to be identified before the dispatch phase or during the commit phase. On the one hand, the identification in the commit phase guarantees that the loop was really executed and is not in a mispredicted path. On the other hand, most of the times the loop would be identified too late and there is no advantage on reusing its instructions anymore. Therefore, the detection in this work is performed in an early phase: after the Instruction Decoding by the SLF block, as illustrated in Figure 2.

After identifying a loop, it has to be buffered in the ROB so that it can be reused. Then, the detection mechanism switches to the second phase. To buffer the instructions, a new flag bit *"Reused"* is introduced in the ROB (see Figure 5). When the instruction has this flag set, the instruction is no longer directly committed from the ROB. Instead, the commit stage starts checking the *Log*. If the instruction path is changed during the buffering, i.e., the branch followed a different path, then the mechanism restarts in the first state and the loop is invalidated.

With one iteration of the loop buffered in the ROB the detection mechanism switches to the third phase. During this new state the instructions are picked from the ROB and the fetch stage is gated-off. The renaming information of the reused instructions is inserted in the *Log*, instead of reinserting these instructions in the ROB.

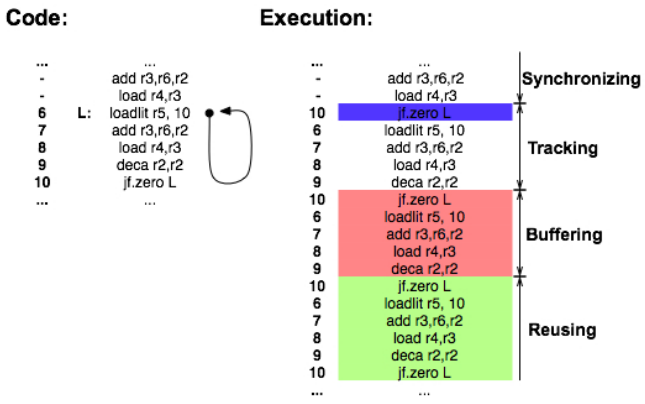

Figure 4: Loop detection example.

While executing sequential iterations of a small loop, a speculative approach is followed until a misprediction appears. When reusing instructions, the dynamic branch prediction is turned off and a static approach is followed instead of the dynamic one used during the normal execution. This works well for most of the loops, since the branches usually tend to maintain the same path of instructions.

When the processor starts reusing instructions a signal is also sent to the fetch stage. The fetch stage, after receiving this signal, updates the PC to the alternative branch path and is turned off. The fetch stage is awaken once more (gated-on) when the signal is turned off. This can happen for two different reasons: *i)* when the loop ends with a misprediction, in this case the fetch stage restarts directly from the new PC allowing a faster restoration of the system's state; *ii)* due to a misprediction from a previous branch or other type of fault, in this case a normal restoring procedure is performed. Both cases imply that the detection mechanism transits again to the first state. If, after a misprediction, there are still active instructions in the *Log* structure waiting for commitment, the detection mechanism is neither allowed to transit from the first phase nor to buffer a new loop.

4.3 The Log Structure

During the reuse phase the instructions are not reinserted in the ROB (saving power). In fact, only the new information from the *Rename Stage* is stored in a smaller *Log* structure. The *Log* structure keeps the physical allocated registers identification, because this information has to be used later in the commit stage or in case of a misprediction.

Between two iterations of a small loop it is possible to have anti-, output and data dependencies that are solved with renaming, in-order issuing and in-order commit. When the loop is unrolled, the instructions pass through the renaming stage, where the operands physical locations are read from the *register alias table* (RAT) and a position is allocated to new data. Reusing the instructions does not affect this procedure, since the information about the renaming of each instruction is stored in the *Log* structure. When executing a loop it is also necessary to guarantee that all the instructions in each iteration were executed before committing. This is guaranteed using a counter for each iteration. Each instruction has a tag with size, in bits, equal to the logarithm of the maximum number of iterations. This tag is also used for validation when a misprediction occurs (section 4.1), and at the end of the execution the counter corresponding to its iteration is incremented. Whenever the counter reaches the number of instructions in the loop, the iteration is finished and can be committed. Stores and loads are executed using a separated queue to allow speculation.

The *Log* structure can be organized as a matrix, as depicted in Figure 6(a), or along only one dimension, as represented in Figure 6(b). In both cases, the *Log* is able to store a maximum number of iterations ("log_y_max") for loops with a maximum number of instructions ("log_x_max"). If a loop is detected with more instructions than this limit, then it is not reused. The fixed sized matrix (structure with two dimensions) is the straightforward way to implement this structure. However, if a loop with a small number of instructions is reused a lot of space in the structure is wasted. The 1D approach allows a more flexible and efficient usage of the structure, because the amount of iterations handled

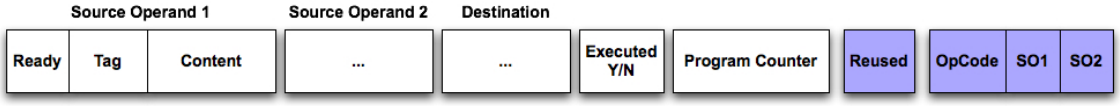

Figure 5: Fields added to a ROB entry.

by the *Log* ("log_x_max") is modified according to the size of the loop.

With the 1D structure, the number of handled iterations (*"log_y_max"*) also depends on the number of counters available to use, since the maximum number of iterations is now variable. The number of counters is equal to the maximum number of iterations in a loop and the size of each counter (in bits) is at least the logarithm (base 2) of the maximum number of instructions in a loop. If the *min_loop_size* (the minimum number of instructions in a loop) is very small (less than 4 instructions), the number of counters is huge and the structure turns out to be inefficient. To use both characteristics of this unidimensional structure, a suitable *min_loop_size* has to be chosen, and only a discrete number of sizes considered, up to *log_x_max*. If the considered sizes are powers of two, from the minimal number chosen, larger counters can be built based on smaller ones, reducing the cost. For instance, counters of modulo 4 and 16 can be built with 2 counters of 2 bits, enabling to accommodate 2 iterations for loops with at most 4 instructions and 1 iteration for loops up to 16 iterations. Therefore, the total cost of these counters is similar to the cost of the counters used for the

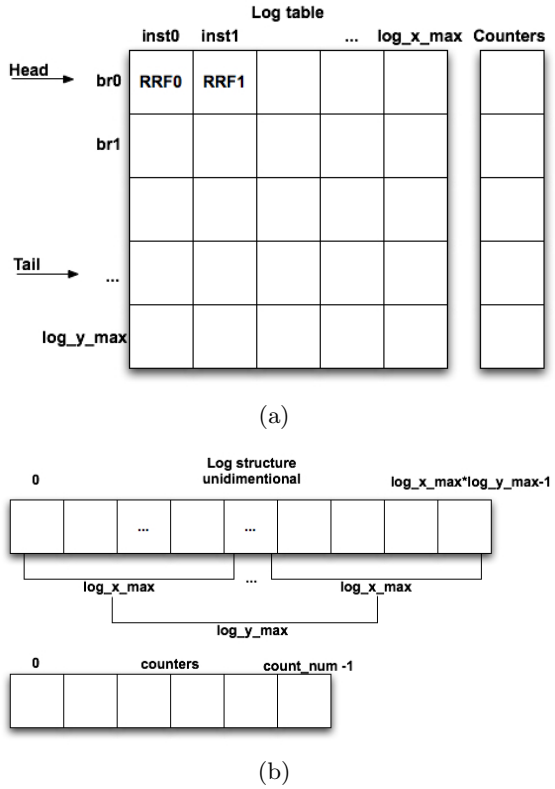

Figure 6: The *Log* structure: (a) 2-D organization; (b) 1-D organization.

two-dimensional structure. With this assumption the maximum number of iterations (*mni*) in the one dimensional structure is calculated by:

$$\text{mni} = \left\lfloor \frac{log\_x\_max \times log\_y\_max}{min\_loop\_size} \right\rfloor \quad (1)$$

4.4 Size Analysis

Although there is no closed form solution for computing the optimal size of the *Log*, a reasonable size for the structure has to be precomputed independently of the application. The ideal size of the structure should be such that the processor never stops due to the lack of space in the *Log*. To do so, it has to handle all the instructions executing on-the-fly, plus the instructions in the reservation stations and the ones waiting for commitment. This also depends on the number of physical registers, but we can consider that all the registers needed are available. With this assumption, it is easy to compute the worst case for the *Log* structure size, when all the instructions can be dispatched except one: the one in the first iteration of the loop and with the highest-possible latency. In this case, the first iteration can not be committed and all the following iterations have to wait in the *Log* structure for commitment until this instruction finishes.

To compute this value, the average latency (*w_lat*) of a load with a miss in the several levels of memory should be used. These latencies are dynamic, depending on the program that is running. In a worst case scenario the application may have inter dependencies between load instructions in different iterations. In order to obtain a reasonable value for the *Log* size, it is used a weighted average between the latency, in cycles, of a memory access and the higher latency of any arithmetic operation.

$$w\_lat = weight_{mem} \times lat_{mem} + weight_{arit} \times lat_{arit} \quad (2)$$

The obtained ideal size (*is*), according to the demonstrations provided in [11], is:

$$is = \left(\left\lceil \frac{w\_lat \times disp\_wd}{log\_x\_max} \right\rceil + 1 \right) \times log\_x\_max \text{ [entries]} \quad (3)$$

The final result obtained should be compared with $\frac{PRF\_size}{log\_x\_max}$ (PRF represents the *Physical Register File*), since it reflects approximately the original number of iterations that can fit into the ROB. The quasi-ideal size should be approximate to the original value in order to keep up the performance due to the loop constraint. The lesser amount of iterations handled by the structure implies the need to wait more time before inserting a new one. Moreover, whenever the loop has too many instructions the time needed until one iteration is committed is higher, which may reduce the performance. On the other hand, a huge structure could prevent power saving. The size of this structure should be carefully selected according to this trade-off, in order to achieve valuable results.

Table 2: Simplescalar baseline configuration

Parameters	Configuration
BTB	1024 sets, 4 way associativity
Branch Predictor	Meta-table, RAS: 4096, 32 entries Combined Predictor (bimodal+2-level) Bimodal: 4096 entries PAg: l1=1024; l2=1024; 10-bit history
Branch misprediction penalty	2 cycles
L1 I-cache L1 D-cache L2 unified cache	32KB, 32B blocks, 1 way assoc., 1 cycle, LRU 256KB, 32B blocks, 8 way assoc., 1 cycle, LRU 1024KB, 64B blocks, 8 way assoc., 6 cycles, LRU
Memory	97 cycles for first chunk, 1 cycle the rest
TLB	ITLB: 16 sets, 4-way DTLB: 32 sets, 4-way 4KB page size, 30 cycle penalty, LRU
Fetch queue size	8 entries
Fetch/Decode/Issue and Commit Width	4 instructions
RUU size	128 entries
LSQ size	80 entries
Function Units	4IALU, 1IMULT, 4FPALU, 1FPMULT Worst case latency: 24 cycles

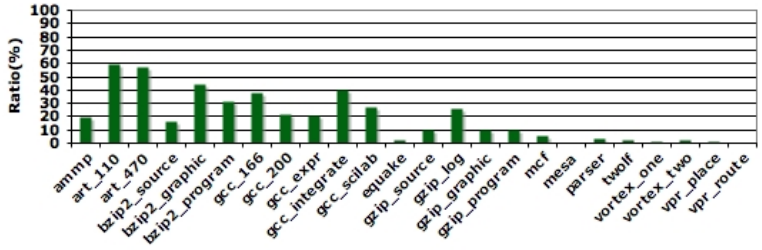

Figure 7: Ratio between the number of instructions reused and the total number of executed instructions.

5. PERFORMANCE EVALUATION

This section describes the experimental setup used during the simulations and presents the results obtained for power and performance.

5.1 Setup

The *Simplescalar* and *Wattch* tools have been used in order to evaluate the performance of the proposed CLU IR method. The *SPEC CPU2000* benchmarks [5] were selected since we are interested in general purpose computing and also because these benchmarks are developed from actual real user applications allowing to measure the performance of the processor.

The *Simplescalar v3.0* toolset [2] simulates an architecture closely related with the MIPS (MIPS-IV ISA [12]); it is a suite of powerful execution-driven computer simulation tools. In this work we used the *sim-outorder* tool from *Simplescalar v3.0*, which is the detailed, out-of-order issue, superscalar processor simulator with support for non-blocking caches and speculative execution. The *Wattch v1.02* [1] software tool was also used to implement a power evaluation methodology within the Simplescalar tool set. It was used in this work in order to analyze the power dissipation for all the major units of the processor, namely the ROB and the *Log*. The load capacitances considered in the Wattch are calculated with the assistance of the CACTI tool, according to suppositions similar to the ones used by *Jouppi et al.* [9, 17].

Since simulating the complete execution of the benchmark suite is a very time consuming task, the *SimPoint* tool [3] was applied to reduce simulation time without introducing significant errors.

Table 2 presents the configurations used as baseline during the simulations. Both the Simplescalar and the Wattch were modified to include the SLF and the *Log* structures. To correctly analyze the effects of the CLU method, several configurations of the *Log* structure were considered during the simulations. These configurations use *Log* size, closer to the values obtained from the empirical analysis presented in section 4.4. In this work we present only the most relevant results for a *Log* structure with a size of *16x12* ($log\_x\_max \times log\_y\_max$), i.e., a *Log* with about 1.5kbit. From the performed simulations, we have concluded that this is the most balanced configuration.

5.2 Obtained Results

The simulation results are normalized with respect to the baseline configuration described in Table 2 for the superscalar microarchitecture. In the first phase, each test was

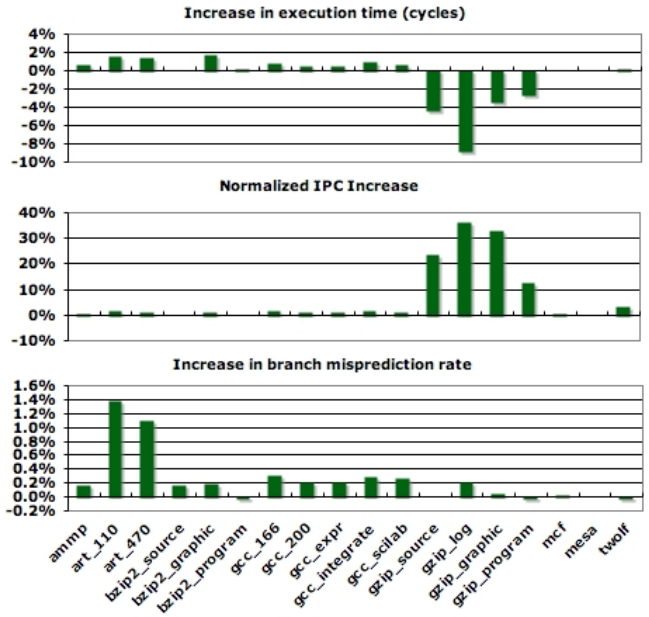

Figure 8: Performance Related Results.

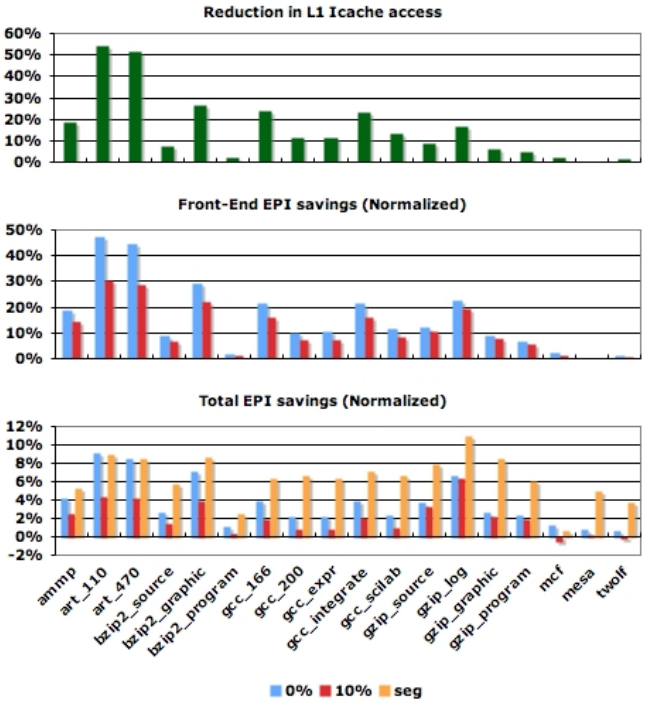

Figure 9: Energy Per Instruction Results.

simulated only once for a *Log* structure reusing loops with a maximum of 16 instructions. The ratio of reusable instructions in each benchmark program, obtained by simulation, is presented in Figure 7.

From the results presented in Figure 7, in some benchmarks there is a large potential for instruction reuse; as an example, the *art_110* potentially allows to reuse about 60% of the instructions. On the other hand, the tests *equake, parser, vortex, vpr, mesa* and *twolf* present lower ratio of reusable instructions (less than 5%). However, these tests can be of great importance: they allow to assess the impact on the processor performance when the proposed technique can not be used. Based in the simulation time, we use the tests *mesa* and *twolf* to perform this evaluation.

Figure 8 presents the performance results when the proposed CLU technique is used in a *Log* structure with 1.5kbit. The presented values include the impact in the overall time execution (in cycles), in the fraction of IPC and in the branch misprediction rate. As it can be seen, the increase in the branch misprediction rate is less than 2%, showing that the static approach for the branch prediction does not have a significant impact in the performance. On the other hand, the IPC slightly increases for all benchmarks, except for *gzip*, where it has a significant increase. Finally, we can also observe that the execution time, in cycles, does not increase more than 2% and has a reduction of up to 8% for the *gzip* simulations. This is an interesting result due to the shorter ROB-based path delivering the instructions faster into the execution, and thus increasing the number of instructions issue in each cycle (IPC gorws more that 20%). Therefore more ILP is exploited and less cycles are required to execute a task. Consequently, the performance of the processor is almost not affected, and sometimes it is even improved.

Power consumption results were obtained with the Wattch tool, together with the number of accesses reported by the simplescalar out-of-order simulator. These results are also used to evaluate the power consumption presented by the additional *Log* structure that we have proposed. In this section the results are shown in relative percentages. For each clock cycle in which an architecture module is not accessed, the Wattch tool considers 0% (ideal) and 10% (non ideal) of the power consumption for the module. In the later case, the tool can also provide results when the ROB is considered to be segmented in 4 parts.

Figure 9 presents the power related results. Such values include the reduction in the L1 I-cache accesses, the front-end *energy per instruction* (EPI) savings and the total EPI savings. As it can be observed from the figure, in most of the tests at least 10% of the cache accesses can be redirected to the ROB, enabling significant energy savings achieved through gating-off instruction fetch and decode units. The most significant energy savings in the front-end occur for the *art* test. However, the results obtained for the total EPI show that not only the *art* but also the *gzip* provide high results for the reduction of the power consumption (in both tests, the saving is more than 8%). As it was expected, tests with fewer loops, such as *mcf, mesa* and *twolf*, present lower EPI savings.

6. CONCLUSIONS

This work proposes a new *constrained loop unrolling* (CLU) method and discusses the functional and structural implications that arise from its implementation in a superscalar processor. Two new hardware structures are included in the baseline superscalar microarchitecture: a controller and a

small *Log* buffer to store important information when instructions are reused.

The simulations performed using the SPEC CPU200 benchmark suite show that the instruction reuse CLU method can reduce the power consumption and improve the efficiency of superscalar processors. Simulation results reveal that there is an average improvement in the power efficiency of the processor for the considered configuration of more than 6% and a peak gain of 10% for the *gzip_log* program.

Better results are expected for applications with regular processing, such as signal processing, data compression, and multimedia as perceived from the good results obtained with the benchmarks *art, bzip2* and *gzip*. Moreover, the obtained results show that the power consumption in the front-end of the processor is reduced by 15%, in average, when the CLU method is applied. As a final conclusion, it can be stated that the design of the *Log* structure is an important step when implementing the proposed CLU method, since its characteristics affect the overall power consumption and performance of the processor.

7. ACKNOWLEDGMENTS

This work has been supported by the HiPEAC European Network of Excellence.

8. REFERENCES

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pages 83–94, June 2000.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[3] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. *The Journal of Instruction-Level Parallelism*, 7, Sep 2005.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003.

[5] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[6] J. S. Hu, N. Vijaykrishnan, S. Kim, M. Kandemir, and M. J. Irwin. Scheduling reusable instructions for power reduction. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, page 10148. IEEE Computer Society, February 2004.

[7] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, 00:132–141, 1998.

[8] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86:82–85, Jan 1998.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 206–218, 1997.

[10] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, July 1997.

[11] F. Pratas. Low power microarchitecture with instruction reuse. Technical Report 21, INESC-ID, September 2007.

[12] C. Price. *MIPS IV Instruction Set, revision 3.2*. MIPS Technologies, Inc., Mountain View, CA, September 1995.

[13] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Companies, Inc., New York, NY, USA, 1st edition, 2005.

[14] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 194–205, June 1997.

[15] E. Talpes and D. Marculescu. Power reduction through work reuse. In *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 340–345, 2001.

[16] E. Talpes and D. Marculescu. Execution cache-based microarchitecture power-efficient superscalar processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(1):14–26, 2005.

[17] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, may 1996.

[18] C. Yang and A. Orailoglu. Power-efficient instruction delivery through trace reuse. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 192–201, Sept. 2006.