Appeared in *Journal of Systems Architecture*, Volume 55, Issue 3, pp. 188-195, 2009, ISSN 1383-7621

Exploiting Selective Instruction Reuse and Value Prediction in a Superscalar Architecture

Arpad Gellert, Adrian Florea, and Lucian Vintan
Computer Science Department, “Lucian Blaga” University of Sibiu, Emil Cioran Street, No. 4, 550025 Sibiu, Romania
{arpad.gellert, adrian.florea, lucian.vintan}@ulbsibiu.ro

Abstract: In our previously published research we discovered some very difficult to predict branches, called unbiased branches. Since the overall performance of modern processors is seriously affected by misprediction recovery, especially these difficult branches represent a source of important performance penalties. Our statistics show that about 28% of branches are dependent on critical Load instructions. Moreover, 5.61% of branches are unbiased and depend on critical Loads, too. In the same way, about 21% of branches depend on MUL/DIV instructions whereas 3.76% are unbiased and depend on MUL/DIV instructions. These dependences involve high-penalty mispredictions becoming serious performance obstacles and causing significant performance degradation in executing instructions from wrong paths. Therefore, the negative impact of (unbiased) branches over global performance should be seriously attenuated by anticipating the results of long-latency instructions, including critical Loads. On the other hand, hiding instructions’ long latencies in a pipelined superscalar processor represents an important challenge itself. We developed a superscalar architecture that selectively anticipates the values produced by high-latency instructions. In this work we are focusing on Multiply, Division and Loads with miss in L1 data cache, implementing a Dynamic Instruction Reuse scheme for the MUL/DIV instructions and a simple Last Value Predictor for the critical Load instructions. Our improved superscalar architecture achieves an average IPC speedup of 3.5% on the integer SPEC 2000 benchmarks, of 23.6% on the floating-point benchmarks, and an improvement in energy-delay product (EDP) of 6.2% and 34.5%, respectively. We also quantified the impact of our developed Selective Instruction Reuse and Value Prediction techniques in a simultaneous multithreaded architecture (SMT) that implies per thread Reuse Buffers and Load Value Prediction Tables. Our simulation results showed that the best improvements on the SPEC integer applications have been obtained with 2 threads: an IPC speedup of 5.95% and an EDP gain of 10.44%. Although, on the SPEC floating-point programs, we obtained the highest improvements with the enhanced superscalar architecture, the SMT with 3 threads also provides an important IPC speedup of 16.51% and an EDP gain of 25.94%.

Keywords: superscalar architecture, SMT architecture, dynamic instruction reuse, load value prediction, speculative execution, power consumption

1. Introduction

In our previous work, we show that unbiased branches are characterized by low prediction accuracies (at average about 70%), irrespective of the prediction information type used in the state-of-the-art branch predictors [Vin06, Gel07]. Since the overall performance of modern superscalar processors is seriously affected by misprediction recovery, these difficult branches represent a source of important performance penalties. As we pointed out in [Gel06], 28.68% of branches are dependent on critical Load instructions (Loads with miss in the L2 data cache that reach the head of the ROB), and 5.61% are unbiased and dependent on a previously committed critical Load instruction. Furthermore, 21.48% of branches depend on MUL/DIV instructions whereas 3.76% are unbiased and depend on MUL/DIV instructions. These dependences involve high-penalty mispredictions becoming serious performance obstacles and causing significant performance degradation in executing instructions from wrong paths. Therefore, the negative impact of unbiased branches, over the global performance should be seriously attenuated by anticipating the results of long-latency instructions. On the other hand, hiding instructions' long latencies in a pipelined superscalar processor represents an important challenge itself.

Our main objective is to develop a superscalar architecture that selectively anticipates the values produced by high-latency instructions. We will focus on Multiply, Division and Loads with miss in the L1 data cache. The reusability degree of MUL and DIV instructions, measured with an unlimited Reuse Table, was 28.9% on the integer benchmarks and 61.9% on the floating-point benchmarks. These instructions would be solved by a Dynamic Instruction Reuse scheme. The reusability degree of Load values was 77.4% on the integer benchmarks and 76.4% on the floating-point benchmarks [Gel08]. However, an additional Reuse Buffer for Load Value (Data) Reuse is not necessary, because a similar reuse mechanism is already provided by the existing L1 and L2 data caches. Therefore, the Load instructions with miss in the L1 data cache (selective approach) would be solved through value prediction.

As a final objective of our research, we quantify the impact of our developed Selective Instruction Reuse and Value Prediction techniques in a Simultaneous Multithreaded Architecture that involves per thread Reuse Buffers (RB) and Load Value Prediction Tables (LVPT). We measure the IPC and the dynamic power consumption of the proposed SMT architecture by varying the number of threads. Also, we evaluate, for different number of threads, the IPC speedup and the EDP gain of a SMT architecture enhanced with Selective Instruction Reuse and Value Prediction against a classical SMT architecture.

The organization of the rest of this paper is as follows. In Section 2, we review related work in the fields of dynamic instruction reuse and value prediction. Section 3 presents the simulation methodology. Section 4 describes the two techniques that we implemented for anticipating the results of long-latency instructions. In Section 5 we illustrate the experimental results obtained using our developed simulator. The last Section suggests directions for future works and concludes the paper.

2. Related Work

Sodani and Sohi in [Sod97] firstly introduced the idea of dynamic instruction reuse. Dynamic instruction reuse is a non-speculative microarchitectural technique that exploits the repetition of dynamic instructions. The main idea is that if an instruction or an instruction chain is reexecuted with the same input values, its output value will be the same. The authors introduced different schemes that maintain the inputs and the results of previously executed instructions in a hardware structure called Reuse Buffer. With instruction reuse, the number of executed dynamic instructions is reduced and the critical path might be compressed. According to the authors' simulations on the SPEC95 benchmarks, at average 26% of dynamic instructions are reusable. This quite high reuse degree is understandable taking into account that less than 20% of the static instructions are generating more than 90% of the repeated dynamic instructions. These useful statistics are qualitatively justified due to the fact that programs are written in a compact (loops, recurrence, inheritance, etc.) and generic manner (the programs have to operate on a variety of data structures). There are some important differences between our approach and Sodani's. We reuse only MUL and DIV instructions and, although we use the same S_v scheme that track operand values for each instruction, our scheme does not require all fields of Sodani's S_v scheme. Since we do not reuse Load instructions, we renounce to the *address* and *memvalid* fields. This reduces the hardware cost with benefits on power consumption, too. Another difference refers to the moment when the instructions are reused: in contrast with Sodani's approach, the Reuse Buffer (RB) is accessed in our architecture during the *issue* stage, because most of the MUL/DIV instructions found in the RB in the *dispatch* stage do not have their operands ready.

Richardson introduced *Instruction Memoization* [Ric93], a technique that consists in storing the inputs and outputs of long-latency operations and reusing the output if the same inputs are encountered again. The memo table is accessed in parallel with the first computation cycle, and the computation halts in the case of hit. Thus, memoing reduces a multi-cycle operation to one-cycle when there is a hit in the memo table. In [Bro00] the authors proposed a memoing technique in order to save power. Brooks et al. used memo tables in parallel with the floating-point and integer multipliers, the floating-point adder, and the floating-point divider. Their experimental results on SPEC92 benchmarks show an average speedup of 1.7% and an average power improvement of 5.4%.

Citron and Feitelson in [Cit02] compare different instruction reuse techniques, including *Instruction Reuse* (IR) and *Instruction Memoization* (IM). The authors splat the Lookup Table into several smaller tables for floating-point instructions, Loads, multi-cycle integer instructions (multiplication and division) and all other single-cycle instructions. Each table contained 256 entries. They used IM only for multi-cycle operations. The evaluation results (reuse degree and speedup) obtained on the SPEC95 benchmarks show that only floating-point applications can benefit from instruction reuse.

Golander and Weiss present in [Gol07] different instruction reuse methods for Checkpoint Processors. In checkpoint microarchitectures a misspeculation initiates the rollback, in which the latest safe checkpoint preceding the point of misprediction is recovered, and the reexecution of the entire code segment between the recovered checkpoint and the mispredicting instruction (selective reissue). The authors proposed two instruction reuse methods for normal execution and other two methods for reexecution after a misprediction. The *Trivial* method identifies trivial arithmetic operations having one of the inputs a neutral element or both operands with the same magnitude. The hardware for detecting trivial computations and selecting the result consists in comparators for the input operands and selectors for the writeback. In our simulator, we implemented the *Trivial* method proposed by Golander. The *SelReuse* method uses a small fully associative reuse cache for long latency arithmetic operations. As the authors are showing, an 8-entry cache is sufficient for reusing most of the available results. The *RbckReuse* method is used for all instruction types from reexecuted paths, excepting control-flow instructions. Finally, the *RbckBr* method is used for the branch instructions from reexecuted paths. The reuse structure maintains only the branch outcome and relies on the BTB for the branch target address. A reuse approach that combines all four methods briefly presented above requires an area of 0.87 mm^2 and consumes 51.6 mW. It achieves average instructions per cycle (IPC) speedup of 2.5% for the SPEC 2000 integer benchmarks, of 5.9% for the SPEC 2000 floating point benchmarks, and an improvement in energy-delay product of 4.80% and 11.85%, respectively.

Lipasti et al. [Lip96] firstly introduced Value Locality as the third facet of the statistical locality concepts used in computer engineering. They defined the value locality as "the likelihood of the recurrence of a previously-seen value within a storage location inside a computer system". Measurements using SPEC95 benchmarks show that value locality on Load instructions is about 50% using a history of one (producing the same value like the previous one) and 80% using a history of 16 previous instances. Based on the dynamic correlation between Load instruction addresses and the values the Loads produce, Lipasti et al. proposed a new data-speculative micro-architectural technique entitled *Load Value Prediction* that can effectively exploit value locality. Load value prediction is useful only if it can be done accurately since incorrect predictions can lead to increased structural hazards and longer Load latency. Starting by Loads' dynamic behavior and classifying them separately (unpredictable, predictable and constants), it can be extracted the full advantage of each case. It can be avoided the cost of mispredictions by detecting the unpredictable Loads and the cost of memory access through identifying highly predictable Loads. An important difference between our value prediction approach and Lipasti's is that we selectively predict Load instructions predicting only those generating a miss in L1 cache. Thus, we attenuate the mispredictions cost and reduce the hardware cost of the speculative micro-architecture. Moreover, since less hardware is required, there is also less power consumption.

Mutlu et al. presented in [Mut06] a new hardware technique named *address-value delta (AVD) prediction*, able to parallelize dependent cache misses. They observed that some Load instructions exhibit stable relationships between their effective addresses and data values, due to the regularity of allocating structures in the memory by the program, which is sometimes accompanied by the regularity in the program's input data. In order to exploit these regular memory allocation patterns, the authors proposed an AVD structure that maintains the Load instructions having a stable address-value difference (delta). Each entry of the AVD table consists in the following fields: *Tag* (the upper bits of the Load's PC), *AVD* (the address-value delta corresponding to the last occurrence of that Load) and *Conf* (a saturating counter that records the confidence of AVD). The *Conf* field is used to avoid predictions for Loads with an unstable AVD. If a Load instruction having a stable AVD occurs with a cache miss, its data value is predicted by subtracting the stable delta from its effective address. This prediction enables the preexecution of dependent instructions, including Loads with cache miss. The experimental results show that integrating a 16-entry AVD predictor into a *runahead* processor improves the average execution time by 14.3%, but only for pointer-intensive applications.

Liao and Shieh proposed in [Lia02] a new scheme that combines value prediction and instruction reuse. The main idea consists in predicting operand values if they are not available and speculatively reusing instructions if the predicted operands match the values from the Reuse Buffer (RB). Obviously, the correct path must be reexecuted in the case of misprediction. If the operands of an instruction are ready and their values match the value fields of the corresponding RB entry, the result is guaranteed to be correct, and therefore the execution is non-speculative. The simulations on the SPEC95 benchmarks showed that this scheme provides an average speedup of 8.9%.

In [Cha08] the authors proposed a hardware-based method, called *early load*, in order to hide the load-to-use latency (the latency that instructions wait for their operands produced by Load instructions) with little additional hardware costs. The key idea is to make use of the time that instructions are waiting in the instruction queue to load the data early, before the Loads are effectively executed, by pre-decoding instructions during the *fetch* stage. Thus, instead of using previous instances (values) of the current Load instruction Chang et al. are using an earlier executed-instance (value) of the current Load instance. In this way, the chance to be a correct value seems to increase. They use a small table, called Early Load Queue (ELQ) that records Load instructions and the early loaded data. The proposed scheme allows Load instructions to load data from memory before the *execution* stage. Obviously, a detection method assures the correctness of the early operation before the Load enters into the execution stage. If the corresponding ELQ entry is valid in the Load's *dispatch* stage, the execution of the Load instruction is completely avoided and all dependent instructions get the data from the ELQ. Unfortunately this method doesn't work for out-of-order speculative architectures whereas our technique does. Also, it works only for very small instruction queues. The experimental results showed that this scheme can achieve a performance improvement of 11.64% on the *Dhrystone* benchmark and 4.97% on the *MiBench* benchmark suite.

3. Simulation Methodology

We developed a cycle-accurate execution driven simulator derived from the M-SIM simulator [Sha05] supporting the unmodified, statically linked Alpha AXP binaries as well as the power estimation as supplied by the Wattch framework [Bro00]. M-SIM extends the SimpleScalar toolset [Bur97] with accurate models of the pipeline structures, including explicit register renaming, and support for the concurrent execution of multiple threads. We modified M-SIM to incorporate our selective instruction reuse and value prediction techniques in order to measure the relative IPC speedup and relative energy-delay product gain when the results of long-latency instructions are anticipated.

All simulation results are generated on the SPEC 2000 benchmarks [SPEC] and are reported on 1 billion dynamic instructions, skipping the first 300 million instructions. For the superscalar architecture we evaluated six floating-point benchmarks (*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*) and seven integer benchmarks: computation intensive (*bzip*, *gcc*, *gzip*) and memory intensive (*mcf*, *parser*, *twolf*, *vpr*). In SMT mode, the M-SIM runs multiple benchmarks as different threads in parallel. Therefore, we combined benchmarks into groups of 2, 3 or 6 depending on the simulated SMT architecture. Thus, we used {*bzip*, *gcc*}, {*gzip*, *parser*}, {*twolf*, *vpr*}, {*applu*, *equake*}, {*galgel*, *lucas*}, {*mesa*, *mgrid*} for our SMT with 2 threads, {*bzip*, *gcc*, *gzip*}, {*parser*, *twolf*, *vpr*}, {*applu*, *equake*, *galgel*}, {*lucas*, *mesa*, *mgrid*} for the SMT with 3 threads, and {*bzip*, *gcc*, *gzip*, *parser*, *twolf*, *vpr*}, {*applu*, *equake*, *galgel*, *lucas*, *mesa*, *mgrid*} for the 6-threaded SMT. Table 1 presents some important parameters of the simulated architecture:

Execution Latencies	**Execution unit**	**Number of units**	**Operation latency**
	intALU	4	1
	intMULT / intDIV	1	3 / 20
	fpALU	4	2
	fpMULT / fpDIV	1	4 / 12
Superscalarity	*Fetch* / *Decode* / *Issue* / *Commit* width = 4		
Branch predictor	bimodal predictor with 2048 entries		
Caches and Memory	**Memory unit**		**Access Latency**
	4-way associative L1 data cache, 32 KB		1 cycle
	8-way associative unified L2 data cache, 512 KB		6 cycles
	Memory		100 cycles
Resources	**Register File:** 32 INT / 32 FP		
	Reorder Buffer (ROB): 128 entries		
	Load/Store Queue (LSQ): 48 entries		

Table 1. Parameters of the simulated architecture

The power consumption measurements are generated using an 80 nm CMOS technology. Figure 1 presents the structure of the simulator.

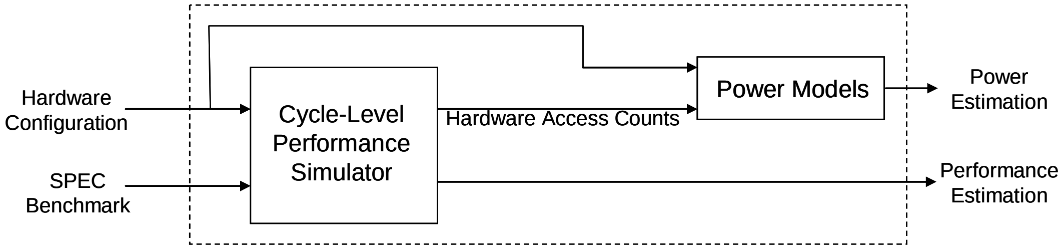

Figure 1. The structure of the simulator

As Figure 1 shows, the simulator generates both performance and power consumption estimation. The detailed power modeling methodology, used in the simulator, is presented in [Bro00]. The dynamic power consumption in CMOS microprocessors is defined as:

$$P_d = C \cdot V_{dd}^2 \cdot a \cdot f \tag{1}$$

where *C* is the capacitance, generated using *Cacti* [Shi01], V_{dd} is the supply voltage, and *f* is the clock frequency. V_{dd} and *f* depend on the assumed process technology. The activity factor *a* indicates how often clock ticks lead to switching activity on average. The power consumption of the modeled units highly depends on the internal capacitances of the circuits. From the capacitance point of view, there are three categories of architectural structures: array structures, content-associate memories, and complex logic blocks. The first two categories are used to model the caches, branch predictors, the reorder buffer, the register renaming table, and the register file, while the last category is used to model functional units.

For the energy measurements, we used the Energy-Delay Product, a widely used metric [Gon96, Bro00, Gol07]:

$$EDP = \frac{Total\ Power}{IPC^2} \qquad (2)$$

The Energy-Delay Product (EDP) represents the processor's total power, divided by the squared IPC.

4. Anticipating Long-Latency Instructions Results

4.1. Selective Dynamic Instruction Reuse

For the MUL and DIV instructions we will use the S_v reuse scheme. The information about instructions is maintained in a direct mapped Reuse Buffer (RB). The RB is accessed during the *issue* stage, because most of the MUL/DIV instructions found in the RB during the dispatch stage do not have their operands ready (91.5% on the integer benchmarks and 64.6% on the floating-point benchmarks). Each RB entry has the following fields: *Tag* (the higher part of the PC), *SV1* and *SV2* (the source values of the MUL/DIV instruction), *Result* (the output value of the MUL/DIV instruction). Since we do not reuse Loads with this scheme, the *address* and *mem valid* fields used in [Sod97] are unnecessary. In this way, our implemented structure is simpler and more cost effective (from hardware budget and power consumption point of view) than the initial scheme proposed by Sodani and Sohi.

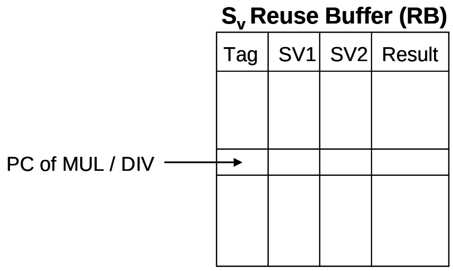

Figure 2. Reuse scheme for MUL & DIV instructions

If a certain MUL/DIV instruction is found in the RB, a reuse test is generated. If the actual operand values, taken from the ROB, match the SV1 and SV2 fields from the selected RB entry, the instruction is not sent to a functional unit, its result value being already available for dependent instructions. Every non-reused MUL/DIV instruction updates the RB in the commit stage: writes the tag, the source values and the result into the corresponding RB entry. From the power consumption point of view, the Reuse Buffer was modeled as a cache array structure using the same power models that the other array structures use. Obviously, the main benefit of reusing long-latency instructions consists in unlocking dependent instructions (see Figure 3). In Figures 3 and

5, all stages except *Execute* stage are a single cycle length; the *Execute* stage has variable length, depending upon the latency of the executing instruction (see Table 1).

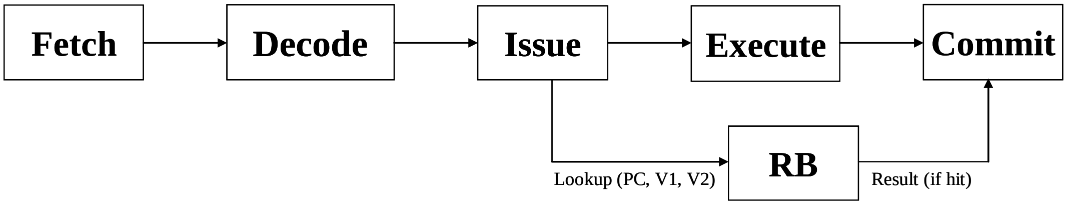

Figure 3. Pipeline with Reuse Buffer (RB)

We also detected trivial operations implementing a technique firstly introduced in [Ric93] by Richardson. We considered the following operations: V*0, V*1, 0/V, V/1 and V/V. A simple hardware scheme for detecting trivial computations and selecting the result is presented in [Gol07] and consists in comparators for the input operands and selectors for the write-back. If during the *dispatch* stage, a MUL instruction is detected with an operand value of 0 or 1, the result is provided by the detector, avoiding the functional unit allocation and execution. In the same manner, if a DIV instruction is detected with the first operand being 0, the second operand 1, or with identical operands, the result is provided by the detector being thus available at the end of the dispatch stage. The Reuse Buffer is accessed during the *issue* stage for the reuse test only if the MUL/DIV operation is not detected as being trivial in the *dispatch* stage.

4.2. Selective Load Value Prediction

We will integrate into our architecture a simple Last Value Predictor used only for Loads with miss in the L1 data cache (selective approach). In this way, the implemented structure is more efficiently used; the collisions number will be lower against the approach that predicts all Load instructions, having tables of the same size. The information about Load instructions is maintained in a direct mapped Load Value Prediction Table (LVPT). The LVPT is accessed during the *issue* stage, only if the current Load instruction involves a miss in the L1 data cache (critical Load). Each LVPT entry has the following fields: *Tag* (the higher part of the PC), *Counter* (a 2-bit saturating confidence counter with two *unpredictable* and two *predictable* states), and *Value* (the Load instruction's result).

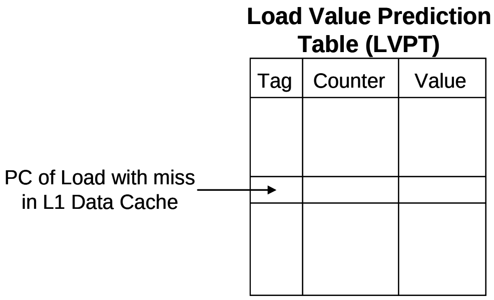

Figure 4. The Last Value Predictor architecture

In the case of a hit in the LVPT, the corresponding *Counter* is evaluated. If the confidence counter is in an unpredictable state, the Load is executed without prediction. Otherwise the *Value* from the selected LVPT entry is speculatively forwarded to the dependent instructions. In the *commit* stage, when the real value is available, in the case of misprediction, a recovery is necessary in order to squash speculative results and selectively re-execute the dependent instructions with the correct values (see Figure 5). This selective reissue_requires a mechanism for propagating misprediction information through the data flow graph to all dependent instructions. We considered in our simulations value prediction latency of one cycle and, in the misprediction case, a recovery taking 7 cycles.

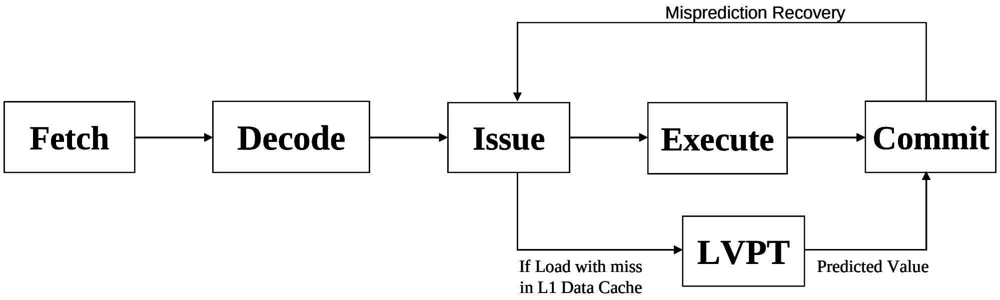

Figure 5. Pipeline with Load Value Predictor

During the *commit* stage, every critical Load updates the LVPT: only the *Counter* field in the case of correct prediction or the *Value* and the *Counter* fields in the case of misprediction. In the miss case the LVPT, the *Tag* and the *Value* are inserted into the selected entry, and the *Counter* is reset (strongly unpredictable state). From the power consumption point of view, the LVPT was modeled as a cache array structure using the same power models that the other array structures use.

5. Experimental Results

Figure 6 presents the reuse degrees obtained with and without detecting trivial operations in the superscalar architecture.

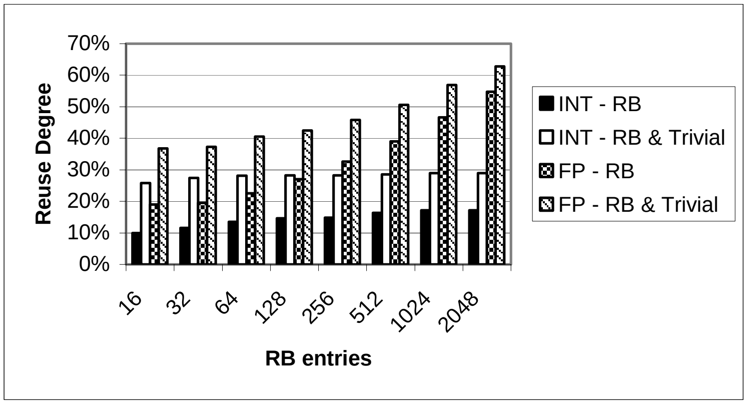

Figure 6. Reuse degrees obtained for different RB sizes with and without trivial operation detection in the superscalar architecture

The RB provides on the integer benchmarks a reuse degree of 17.2% with an RB of 1024 entries, compared with the reusability degree of 28.9% (the upper limit obtained with an unlimited RB). It was more efficient for the floating-point benchmarks, where we obtained a reuse degree of 54.8% with an RB of 2048 entries, compared with the reusability degree of 61.9% (through an unlimited RB). As Figure 6 shows, trivial operations detection improves significantly the reuse degree.

Table 2 presents the reuse degrees, the IPC, and the power consumption obtained with the superscalar architecture, on the integer and floating-point SPEC 2000 benchmarks, by using the S_v reuse scheme together with the Trivial Operation Detector for the MUL and DIV instructions. The *Reuse Degree* columns represent the percentage of reused MUL and DIV instructions across all the evaluated integer and floating-point benchmarks. The *IPC* represents the average instructions per cycle. The *RB Power* column shows the additional dynamic power dissipated by the RB for each evaluated size in *mW* and in percentages reported to the total processor power.

RB entries	SPEC2000 integer		SPEC2000 floating-point		RB Power	
	Reuse Degree [%]	IPC	Reuse Degree [%]	IPC	[mW]	[%]
0 (no RB)	–	1.6857	–	2.0410	0	0.000
16	25.8	1.6881	36.8	2.0612	7.2	0.008
32	27.4	1.6862	37.3	2.0613	12.7	0.014
64	28.1	1.6862	40.5	2.0747	16.3	0.018
128	28.2	1.6862	42.5	2.0752	28.8	0.031
256	28.2	1.6862	45.8	2.0787	38.4	0.042
512	28.5	1.6862	50.6	2.0828	70.2	0.077
1024	29.0	1.6862	56.9	2.0863	99.6	0.109
2048	29.0	1.6862	62.8	2.0888	178.8	0.195

Table 2. Reuse degree, IPC and power consumption obtained with the superscalar architecture using the RB and Trivial Operation Detector on the SPEC2000 benchmarks

The very low IPC gain measured on the integer benchmarks is justified because only about 11 million instructions were reused from a total of 7 billion across all the integer benchmarks. Moreover, reusing MUL/DIVs belonging to wrong speculated paths frequently involves issuing some long latency Loads. These critical instructions would not be executed without successful reuse.

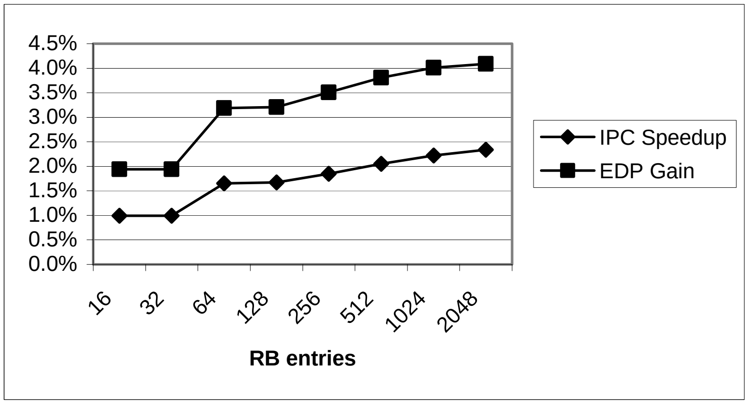

Figure 7. Relative IPC speedup and relative energy-delay product gain on the floating-point benchmarks with RB and Trivial Operation Detection in the superscalar architecture

Although the RB structure dissipates additional dynamic power, reusing long-latency instructions increases the IPC and therefore lowers the total energy consumption (see Figure 7). We determined the energy-delay product for the superscalar architecture without RB and for the architecture with RB of different sizes. The *EDP Gain* represents the relative energy-delay product improvement for each RB size.

The speedup is insignificant in the case of the integer benchmarks, due to the significantly lower number of MUL and DIV instructions. Consequently, the energy-delay product is better only for RB sizes between 16 and 128 entries, but the improvement is insignificant. These results are in concordance with Citron [Cit02] who also remarked the poor evaluation results (reuse degrees and speedups) obtained on the SPEC95 integer benchmarks. Therefore a significant benefit of MUL/DIV instructions reuse is achieved only for floating-point applications.

Table 3 presents the prediction accuracy, the IPC, and the power consumption obtained by evaluating our developed superscalar architecture with MUL/DIV Reuse Buffer of 1024 entries and Trivial Operation Detector for the MUL and DIV instructions and with Last Value Predictor for critical Load instructions. The *PA* columns represent the prediction accuracy of critical Loads. The *IPC* represents the average instructions per cycle. The *LVPT Power* column shows the additional dynamic power dissipated by the LVPT for each evaluated size in *mW* and in percentages reported to the total processor power.

LVPT entries	**SPEC2000 integer**		**SPEC2000 floating-point**		**LVPT Power**	
	PA	**IPC**	**PA**	**IPC**	**[mW]**	**[%]**
0 (no RB, LVP)	–	1.6857	–	2.0410	0	0.000
16	94.0	1.7066	99.7	2.1873	6.4	0.007
32	93.5	1.7094	99.8	2.2333	8.7	0.009
64	92.6	1.7245	99.8	2.3533	14.6	0.016
128	91.0	1.7318	99.7	2.3915	19.9	0.022
256	88.7	1.7351	99.5	2.4378	33.6	0.037
512	88.1	1.7387	99.3	2.4484	48.0	0.052
1024	87.1	1.7456	99.2	2.5241	84.9	0.092
2048	87.2	1.7460	99.1	2.5320	128.1	0.139

Table 3. Prediction accuracy, IPC and power consumption obtained with the superscalar architecture using an RB of 1024 entries, the Trivial Operation Detector and the LVPT

Figure 8 presents the relative IPC speedup and the relative energy-delay product improvement for the integer and floating-point benchmarks. We determined the energy-delay product for the superscalar architecture without RB and LVPT and for the architecture with a RB of 1024 entries and LVPTs of different sizes. The *EDP Gain* represents the relative energy-delay product improvement for each LVPT size. As it can be observed, the optimal LVPT size is 1024. Both IPC speedup and EDP gain are significantly higher on the floating-point benchmarks compared to the integer benchmarks. This difference occurs because the number of critical Loads is more than twice higher in the floating-point benchmarks. The difference is further accentuated by the percentage of predicted critical Loads (classified as predictable by LVPT confidence counters) which is 85% on the floating-point benchmarks and only 40% on the integer benchmarks [Gel08]. Finally, the difference is also slightly increased by the higher prediction accuracy obtained on the floating-point benchmarks.

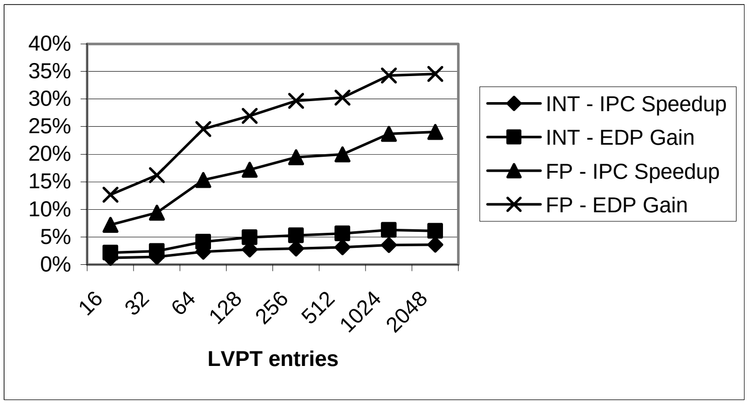

Figure 8. Relative IPC speedup and relative energy-delay product gain with the superscalar architecture using a Reuse Buffer of 1024 entries, the Trivial Operation Detector, and the Load Value Predictor

We also measured the memory traffic reduction as the percentage of correctly predicted Loads reported to the total number of memory accesses. Our evaluations show an average memory traffic reduction of 1.58% on the integer benchmarks and of 10.93% on the floating-point benchmarks, which are in concordance with our energy consumption estimations.

The selective instruction reuse approach proposed by Golander and Weiss (presented in paragraph 2) achieves an average IPC speedup of 2.5% on the SPEC 2000 integer benchmarks, of 5.9% on the floating point benchmarks, and an improvement in energy-delay product of 4.80% and 11.85%, respectively. In comparison, our improved superscalar architecture achieves an average IPC speedup of 3.5% on the integer SPEC benchmarks, 23.6% on the floating-point benchmarks, and an improvement in energy-delay product of 6.2% and 34.5%, respectively.

As a final objective of this research, we quantified the impact of our developed techniques for anticipating long-latency instructions results in a simultaneous multithreaded architecture that involves per thread RBs and LVPTs. We measured the IPC and the dynamic power consumption of the proposed SMT architecture by varying the number of threads. Figure 9 presents the IPC obtained by evaluating our developed superscalar and SMT architectures with and without Reuse Buffer and Load Value Predictor. According to our previous results, we optimally sized the RB and the LVPT to 1024 entries. The RB and LVPT structures improve the IPC on all the evaluated general-purpose applications within all architectural configurations (superscalar and SMT). Therefore, we consider that the Worst Case Execution Time (WCET) is not increased by our proposed techniques. As far as concern floating-point benchmarks, the highest improvement was obtained with one thread, and as the number of threads grows, the IPC improvement becomes lower. With fewer threads, the ten shared functional units (see Table 1) are underused and therefore the Selective Instruction Reuse and Value Prediction techniques have an important improvement potential. With a higher number of threads, the same ten functional units are highly used by the SMT engine, thus both the instruction reuse and value prediction mechanisms becoming less important. Therefore, especially on floating-point benchmarks, with six threads we obtained the best IPC but the lowest relative IPC speedup.

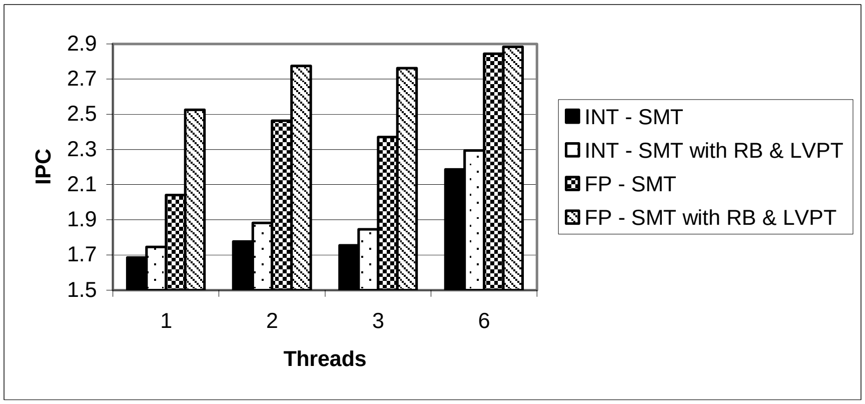

Figure 9. IPC obtained using the SMT architecture with and without RB & LVPT on the SPEC 2000 benchmarks

Finally, we evaluated, for different number of threads, the IPC speedup and the EDP gain of a SMT architecture enhanced with Selective Instruction Reuse and Value Prediction against a classical SMT architecture. In Figure 10 the first and third bars represent the EDP gains obtained with our superscalar (one thread) and SMT architecture (2, 3 and 6 threads) on the floating-point and integer benchmarks, respectively. The second and fourth bars presents the IPC speedups achieved with the same architectures.

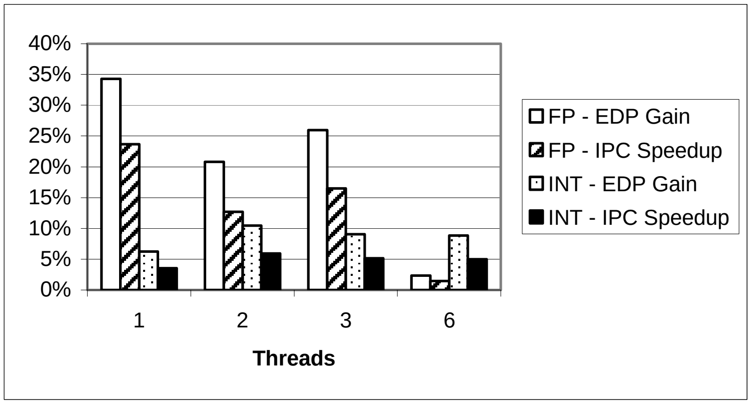

Figure 10. Relative IPC speedup and EDP gain (enhanced SMT vs. classical SMT) by varying the number of threads

As Figure 10 depicts, the RB and LVPT structures achieved IPC speedups and EDP gains on all the simulated configurations. The best improvements on the integer benchmarks have been obtained with 2 threads: an IPC speedup of 5.95% and an EDP gain of 10.44%. Although, on the floating-point benchmarks, we obtained the highest improvements with the enhanced (LVPT + RB) superscalar architecture, the SMT with 3 threads also provides an important IPC speedup of 16.51% and an EDP gain of 25.94%. Analyzing Figure 9 we can observe the advantage of SMT architectures against the superscalar architecture irrespective these are enhanced or not with selective instruction reuse and value prediction mechanism.

6. Conclusions and Further Work

In this study, we have presented and evaluated a superscalar architecture that selectively anticipates the values produced by high-latency instructions. We developed a Reuse Buffer and a Trivial Operation Detector for MUL and DIV instructions and a Last Value Predictor for critical Load instructions, and we integrated all these structures into the M-SIM simulator [Sha05].

The experimental results, performed on the SPEC 2000 benchmarks, show a significant speedup and reduced energy consumption for the proposed architecture. Using a Reuse Buffer of 1024 entries together with the Trivial Operation Detector improves the IPC with over 2.2% and reduces the energy consumption with 4% on the floating-point benchmarks. Predicting critical Load instructions through an additional Last Value Predictor, improves the IPC with 3.5% on the integer benchmarks and with 23.6% on the floating-point benchmarks. This significant speedup lowers the energy consumption with 6.2% on the integer benchmarks and with 34.5% on the floating-point benchmarks, which are far better than the improvements achieved by the selective instruction reuse approach proposed by Golander and Weiss: 4.80% and 11.85%, respectively.

Finally, we have studied the impact of selective instruction reuse and value prediction in a Simultaneous Multithreaded architecture. We used these methods to anticipate the results of long-latency instructions (Mul, Div, Load), as we did within the superscalar architecture. We implemented private RBs and LVPTs for each thread. Our simulation results, performed on the forementioned benchmarks, show that the IPC is better on all evaluated SMT configurations, when the RB and LVPT structures are used. With fewer threads, the shared functional units are underused and therefore the Selective Instruction Reuse and Value Prediction techniques have an important improvement potential. However, as the number of threads grows the IPC speedup decreases, because the shared functional units are better exploited due to the higher thread-level parallelism and therefore the RB and LVPT structures become less important. We measured the highest IPC of 2.29 on the integer and 2.88 on the floating-point benchmarks with our six-threaded enhanced SMT architecture. As a conclusion, applying some well-known anticipatory techniques selectively on long-latency instructions provides serious performance gain and significantly reduces energy consumption in superscalar and even in multithreaded architectures.

As a further work, understanding and quantifying instruction reuse and value prediction benefits in a multicore architecture might be a very important challenge.

Acknowledgments

This work was partially supported by the Romanian National Council of Academic Research (CNCSIS) through the research grants TD-248/2007 and A-39/2007.

References

[Bro00] Brooks D., Tiwari V., Martonosi M., *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*, International Symposium on Computer Architecture, Vancouver, 2000.

[Bur97] Burger D., Austin T., *The SimpleScalar tool set*, version 2.0, Technical Report 1342, University of Wisconsin - Madison Computer Sciences Department, 1997.

[Cha08] Chang S.C., Li W.Y.H., Kuo Y.J., Chung C.P., *Early Load: Hiding Load Latency in Deep Pipeline Processor*, Proceedings of The Asia-Pacific Computer Systems Architecture Conference (ACSAC), Taiwan, August 2008.

[Cit02] Citron D., Feitelson D., *Revisiting Instruction Level Reuse*, Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD), May 2002.

[Gel06] Gellert A., Florea A., *Finding and Solving Difficult Predictable Branches*, Science and Supercomputing in Europe, Barcelona, Spain, 2006.

[Gel07] Gellert A., Florea A., Vintan M., Egan C., Vintan L., *Unbiased Branches: An Open Problem*, Twelfth Asia-Pacific Computer Systems Architecture Conference (ACSAC'07), Seoul, Korea, August 2007.

[Gel08] Gellert A., *Developing and Improving the Performances of Some Predictive Architectures*, Technical Report, "Lucian Blaga" University of Sibiu, April 2008.

[Gol07] Golander A., Weiss S., *Reexecution and Selective Reuse in Checkpoint Processors*, HiPEAC Journal, Vol. 2, Issue 3, 2007.

[Gon96] Gonzalez R., Horowitz M., *Energy Dissipation in General Purpose Microprocessors*, IEEE Journal of Solid State Circuits, Vol. 31, No. 9, September 1996.

[Lia02] Liao C.H., Shieh J.J., *Exploiting Speculative Value Reuse Using Value Prediction*, Seventh Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, February 2002.

[Lip96] Lipasti M. H., Wilkerson C. B., Shen J. P., *Value Locality and Load Value Prediction*, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 138-147, October 1996.

[Mut06] Mutlu O., Kim H., Patt Y. N., *Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses*, IEEE Transactions on Computers, Vol. 55, No. 12, December 2006.

[Ric93] Richardson S., *Exploiting trivial and redundant computation*, 11th Symposium on Computer Arithmetic, July 1993.

[Sha05] Sharkey J., Ponomarev D., Ghose K., *M-SIM: A Flexible, Multithreaded Architectural Simulation Environment*, Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.

[Shi01] Shivakumar P., Jouppi N. P., *Cacti 3.0: An Integrated Timing, Power, and Area Model*, WRL Research Report, Aug 2001, USA.

[Sod97] Sodani A., Sohi G., *Dynamic Instruction Reuse*, The 24th Annual International Symposium on Computer Architecture (ISCA'97), Denver, 1997.

[SPEC] SPEC2000, *The SPEC benchmark programs*, http://www.spec.org.

[Vin06] Vintan L., Gellert A., Florea A., Oancea M., Egan C., *Understanding Prediction Limits through Unbiased Branches*, Eleventh Asia-Pacific Computer Systems Architecture Conference (ACSAC'06), Shanghai, China, September 2006.