Efficient Out-of-Order Execution of Guarded ISAs

NATHANAEL PRÉMILLIEU, ARM Ltd
ANDRÉ SEZNEC, INRIA/IRISA

ARM ISA-based processors are no longer low-cost, low-power processors. Nowadays, ARM ISA-based processor manufacturers are striving to implement medium-end to high-end processor cores, which implies implementing a state-of-the-art out-of-order execution engine. Unfortunately, providing efficient out-of-order execution on legacy ARM codes may be quite challenging due to guarded instructions.

Predicting the guarded instructions addresses the main serialization impact associated with guarded instructions execution and the multiple definition problem. Moreover, guard prediction allows one to use a global branch-and-guard history predictor to predict both branches and guards, often improving branch prediction accuracy. Unfortunately, such a global branch-and-guard history predictor requires the systematic use of guard predictions. In that case, poor guard prediction accuracy would lead to poor overall performance on some applications.

Building on top of recent advances in branch prediction and confidence estimation, we propose a hybrid branch-and-guard predictor, combining a global branch history component and global branch-and-guard history component. The potential gain or loss due to the systematic use of guard prediction is dynamically evaluated at runtime. Two computing modes are enabled: systematic guard prediction use and high-confidence-only guard prediction use.

Our experiments show that on most applications, an overwhelming majority of guarded instructions are predicted. Therefore, a simple but relatively inefficient hardware solution can be used to execute the few unpredicted guarded instructions. Significant performance benefits are observed on most applications, while applications with poorly predictable guards do not suffer from performance loss.

Categories and Subject Descriptors: C.1.3 [Other Architecture Styles]: Pipeline Processors

General Terms: Pipeline processors

Additional Key Words and Phrases: Superscalar processors, branch prediction, predicated execution

ACM Reference Format:

Nathanael Prémillieu and André Seznec. 2014. Efficient out-of-order execution of guarded ISAs. *ACM Trans. Architect. Code Optim.* 11, 4, Article 41 (December 2014), 21 pages.
DOI: <http://dx.doi.org/10.1145/2677037>

1. INTRODUCTION

Most instruction sets offer a limited form of guarded instructions, generally the conditional move, for example, X86, Alpha, MIPS, and SPARC V9. For these instruction sets, the compiler has a limited option to generate if-converted branches [Allen et al. 1983], and in practice, the number of guarded instructions in the generated codes is quite

This work was partially supported by the European Research Council Advanced Grant DAL No 267175. This work was done while Nathanael Prémillieu was with INRIA/IRISA.

Authors' addresses: N. Prémillieu, ARM Ltd., Cambridge, England; A. Seznec, INRIA/IRISA, Rennes, France. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/12-ART41 \$15.00

DOI: <http://dx.doi.org/10.1145/2677037>

limited. The impact of guarded instructions on the effective performance of the processor is also limited. On the other hand, other instruction sets such as ARM-v7 or IA64 have taken a much more radical approach: (nearly) all instructions can be guarded. Therefore, the compiler has much more opportunity to generate guarded instructions.

The ARM ISA now dominates the low-power general-purpose processor segment. With the rise of mobile devices (smartphones, tablets), there is a constant demand for higher performance. While the new ARM-v8 ISA was introduced recently and might dominate the high-end market in a few years, the main demand will remain with low-cost ARM-v7 processors. To achieve high performance with the ARM-v7 ISA, the manufacturers will have to adapt all the concepts that were traditionally reserved to high-end microprocessors, including out-of-order execution. However, providing efficient out-of-order execution on a fully guarded ISA may be quite challenging.¹

The main difficulty for out-of-order execution of guarded instructions is the multiple definition problem [Wang et al. 2001]. This arises when the last instruction that may have written an architectural register R_i was a guarded instruction. In that case, when renaming the registers for a subsequent instruction I that uses R_i as an operand, one has to determine the effective physical register that will provide the value of R_i to instruction I , either the physical destination register of the guarded instruction or the old physical register associated with R_i . A working yet not efficient solution is to insert an extra nonarchitectural instruction after the guarded instruction [Alpha 1999]. This nonarchitectural instruction writes either the result of the operation of the guarded instruction or the old value depending on the dynamic guard (see Figure 4 in Section 2.3). However, this solution may hurt performance as it serializes the execution of possibly independent instructions, for example, when the same register is written on both paths of a branch that has been if-converted, thus reducing the available instruction-level parallelism. More aggressive solutions [Pnevmatikatos and Sohi 1994; Chang et al. 1996; Wang et al. 2001] have been proposed to handle the multiple definition problem, but they induce a significant hardware overhead.

Predicting guarded instructions addresses the multiple definition problem [Chuang and Calder 2003]. However, systematic usage of guard prediction may sometimes lead to a high guard misprediction rate and therefore to poor overall performance. Restricting the use of guard prediction to the high-confidence predictions appears to limit performance degradation [Quiñones et al. 2006], especially when using a perceptron predictor [Quiñones et al. 2007]. Unfortunately, this cannot be transferred to using a state-of-the-art branch predictor such as TAGE [Seznec and Michaud 2006] or GEHL [Seznec 2005] (see Section 5.1).

In this article, we build on recent advances in branch prediction [Seznec and Michaud 2006] and confidence estimation [Seznec 2011a] for efficiently supporting out-of-order execution on a guarded ISA. We propose a hybrid branch-and-guard predictor, combining a global branch history predictor and a global branch-and-guard history predictor. This hybrid predictor will be referred to as the BO-BG predictor, for Branch-Only history/Branch-and-Guard history. The BO-BG predictor is often more accurate on branch prediction than its branch-only history component. However, on some applications or application phases, the guard misprediction rate of BO-BG is quite high. In these cases, systematic use of guard predictions leads to lower performance.

Therefore, we introduce a simple heuristic to dynamically estimate the performance benefit or loss of the systematic usage of guard prediction. Our BoL (for Benefit-or-Loss) heuristic determines whether to run in *systematic guard prediction use mode* or in *high-confidence-only guard prediction use mode*. When running in

¹The handling of guarded instructions in currently implemented out-of-order execution processors is largely undocumented.

Fig. 1. Illustration of the register renaming process in an out-of-order processor.

high-confidence-only guard use mode, the global branch-and-guard history can be corrupted with mispredicted guards; therefore, only the branch-only predictor component is used.

Our experiments using the BO-BG predictor and the BoL heuristic show that, on most applications, most guarded instructions are predicted. Therefore, a simple but relatively inefficient hardware solution can be used to execute the few unpredicted guarded instructions. Compared with out-of-order execution without guard prediction, significant performance benefits are encountered on most applications, while applications with poorly predictable guards do not suffer any from performance loss. Moreover, our experiments also show that an aggressive implementation of guarded instruction execution is not worth the extra hardware complexity and power consumption.

The remainder of this article is organized as follows. Section 2 provides background on the multiple definition problem in an out-of-order execution processor using guarded ISAs. Section 3 presents related works on guarded instructions and guard prediction. Guards in the ARM-v7 ISA are described in Section 4. Section 5 details our BO-BG predictor proposal and the associated BoL heuristic. Section 6 presents our evaluation framework. Section 7 presents our experimental simulation results on ARM codes generated with a standard gcc compiler. Finally, Section 8 concludes this study.

Terminology

When referring to ISAs, guards and predicates are used as synonyms. To avoid repeating “predicate prediction” and “predicted predicate” in the article, we will use the term guard apart in the expression “False Predicated Conditional Move” that was previously coined by Quiñones et al. [2006].

2. EXECUTING GUARDED INSTRUCTIONS ON AN OUT-OF-ORDER ENGINE

2.1. Register Renaming (No Predication)

In an out-of-order execution engine, the mapping table is used to store the links between architectural registers and their associated physical registers value. This mapping table is used to avoid false dependencies between instructions that access the same architectural register. Hence, for each instruction, the rename stage assigns new physical registers to the architectural destination registers and the architectural source registers are renamed. A physical register P associated with architectural register R is considered as dead when the next write to R has been committed; at this time, it can be inserted in the free list and used again for renaming.

Figure 1 illustrates an example of the register renaming process. Instruction I reads from architectural registers $R1$ and $R2$ and writes into architectural register $R3$. To obtain the renamed form of instruction I , one has to read the mapping table. In this

Fig. 2. The multiple definition problem on an out-of-order execution engine.

example, $R1$ is mapped to $P12$ and $R2$ to $P15$. The result register $R3$ is assigned to the first physical register available in the free list of physical register, $P22$ in this case. Then, the renamed form of I is $I : P22 \leftarrow P12, P15$.

All these steps are performed by the renaming stage before executing the instructions. Though renaming is applied to multiple instructions in parallel, the process preserves the in-order semantic of the program.

2.2. The Multiple Definition Problem on Out-of-Execution Processors

When considering a guarded instruction, one cannot determine at the rename stage whether it will effectively write its architectural register target at write-back or not because the guard value is often not known at this point.

Figure 2 illustrates this issue, known as the multiple definition problem [Wang et al. 2001]. I_1 conditionally writes to architectural register $R1$, I_1 being guarded with the guard p . After renaming, I_1 conditionally writes to $P1$. I_2 reads from $R1$, but it is not possible to know whether the correct physical register associated with $R1$ is $P1$ or $P11$ before the guard associated with I_1 is computed.

2.3. Dealing with the Multiple Definition Issue

2.3.1. False Predicated Conditional Moves. On an out-of-order execution processor, the execution of guarded instruction I writing the architectural register Res

$$(guard)?Res \leftarrow Operation(Op1, Op2)$$

should result at execution stage in

$$P_{after} = (guard)?Operation(Op1, Op2) : P_{before},$$

where P_{before} and P_{after} are, respectively, the physical registers assigned to architectural register Res before and after instruction I . That is, if the guard is false, the instruction copies the value from the physical register previously allocated to Res to the newly allocated physical register.

Quiñones et al. [2006] refer to this functionality as *False Predicated Conditional Move* (FPCM).

Direct implementations of FPCM are used in the literature [Pnevmatikatos and Sohi 1994; Chang et al. 1996]. Figure 3 illustrates the artificial dependency that is created by FPCM. Instruction $I2$ must be executed after instruction $I1$, and this, independently of the effective value of the guard.

The direct implementation of FPCM tends to have very significant hardware complexity in the design. Every guarded instruction has an extra physical register operand. Therefore, extra complexity is added in most of the stages of the pipeline, particularly on the physical register file (extra read ports), on the bypass network, and on operand tracking in the issue logic.

Fig. 3. False Predicated Conditional Move; each guarded instruction is added an extra register operand: the old value of the target operand.

Fig. 4. Split FPCM: the first micro-operation unconditionally executes the operation; a second micro-operation selects between the new value or the old value.

2.3.2. Split FPCM. The aforementioned complexity cannot be justified when the use of guarded instructions is quite infrequent, for example, when the instruction set only allows conditional moves.

An alternative implementation consists of detecting the guarded instruction at decode time and splitting the instruction into two consecutive micro-operations: the first micro-operation executing the computation and the second micro-operation selecting between the previous target register value and the result of the first micro-operation. We will refer to this implementation as *split FPCM*.²

The two micro-operations corresponding to the instruction I

$$(guard)?Res \leftarrow Operation(Op1, Op2)$$

are

$$P_{new} = Operation(Op1, Op2)$$

and

$$P_{after} = (guard)?P_{new} : P_{before}.$$

Figure 4 illustrates the serialization of the sequence of accesses on the registers, as well as the artificial creation of long dependency chains. Even if micro-operations I1 and

²Despite extensive search, we have not found any bibliographical reference on split FPCM. However, this technique has been known in the industry since the mid-1990s [Alpha 1999].

$I2$ were executed on the same cycle T , the physical register P8 mapping architectural register R1 for subsequent instructions will not be valid before $I2'$ is executed (cycle $T + 2$ at best). Such an implementation may impair performance when a significant amount of guarded instructions are executed.

2.3.3. Select- μ Operation. The split FPCM mechanism inserts systematically an extra micro-operation for each guarded instruction. This results in a longer latency for guarded instruction as well as systematic serialization of potentially independent instructions.

Wang et al. [2001] propose the select- μ op instruction, a solution to reduce this overhead and limit serialization to cases where it is mandatory. This instruction is conceptually similar to the ϕ -function used in the Single Static Assignment analysis [Cytron et al. 1991]. The select- μ op is inserted just before a multiple definition must be resolved, for example, when a nonguarded instruction reads a register that was last written by a guarded instruction.

Compared with split FPCM, select- μ op has several advantages. First, it postpones the insertion of the selection micro-operation till the effective use of the architectural register with a different guard. That is, in case of successive guarded writes on the same architectural register using the same guard, a single select- μ op selection instruction is inserted. Second, the (speculative) executions of two guarded instructions writing the same register but guarded with opposite guards are not sequentialized and result in a single select- μ op insertion.

However, the select- μ op mechanism is rather complex to implement in hardware; for instance, each entry in the register mapping table must record two different physical register numbers and a physical guard register number. A single guarded instruction with two register operands and one guard can trigger up to three select- μ op insertions (one per register operand and one for the guard). Triggering these insertions requires one to check the mapping table to look for the previous guarded definitions of the registers, adding extra complexity to the renaming stage. In comparison, the treatment of split FPCM can be implemented just at the exit of the decode stage.

Moreover, in their study, Wang et al. [2001] fail to indicate how precise interruptions could be implemented: speculative registers allocated to an instruction may survive forever in the model proposed in Wang et al. [2001].³

3. RELATED WORKS ON BRANCH AND GUARD PREDICTIONS

Efficiently dealing with control instructions in a processor has always been a challenge. Two directions have been proposed: using prediction to know in advance the direction and the target of the branch [Smith 1981] and using guarded instructions.

If-conversion was proposed by Allen et al. [1983]. They define an algorithm to convert control dependencies into data dependencies by replacing branches and their dependent instructions by guarded instructions. The guarded instructions are only executed if their guard is evaluated to true. This conversion algorithm is often called if-conversion. If-conversion allows one to merge the taken and not-taken paths in the binary; it removes a branch and allows one to sequence both paths at the same time. However, it is not possible to if-convert all conditional branches. It is not always performance effective either, since both paths are fetched, increasing occupancy of the processor resources. Thus, one should only if-convert a subset of the convertible branches. Compilers often if-convert short branches only.

Combining branch prediction and guarded execution has been proposed in several studies [Kim et al. 2005, 2006; Chang et al. 1996]. The main idea is to have the

³Solutions to solve this issue are out of the scope of our study.

compiler generating branch instructions and taken and not-taken paths for easy-to-predict control flow while hard-to-predict control flow is treated through if-conversion. This reduces the number of mispredicted branches at runtime and should increase performance. Chang et al. [1996] use profiling to identify the hard-to-predict branches to convert. They show that profiling is efficient at identifying hard-to-predict branches. Kim et al. [2005] further propose the wish branches. For each candidate of if-conversion, two versions of the code are generated, the guarded and the nonguarded code. Then, the executed version is chosen dynamically based on a confidence estimation.

Several studies [Pnevmatikatos and Sohi 1994; Tyson 1994] point out that removing branches by if-conversion may impact the accuracy of branch prediction on the other branches. Simon et al. [2003] observe that the outcomes of some branches can be directly related to the value of some guards, and therefore that applying if-conversion on selected branches often decreases branch prediction accuracy for other branches. They also propose including guard information in the global branch history to try to capture the correlation lost by the if-conversion. However, their approach is limited to include effective guard information when known at fetch time.

Guard prediction solves the multiple definition problem. Chuang and Calder [2003] propose using a guard predictor. The predictor is derived from a branch predictor. Contrary to branch prediction, on a guard misprediction, there is no need to squash the entire pipeline. Therefore, the authors propose a selective replay mechanism, where only the instructions that depend on the mispredicted instruction are re-executed. Quiñones et al. [2006] propose selectively using the guard prediction. All guards are predicted, but the effective use of prediction is triggered on a per-guard basis using a confidence estimator. When the confidence is high enough, the prediction is used. If not, the guarded instruction is handled through False Predicate Conditional Moves. In a later work, Quiñones et al. [2007] identify that branch outcomes as well as guard values are often correlated with former guard values and propose a combined branch-and-guard predictor. The predictor used in Quiñones et al. [2007] is a local history/global branch-and-guard history perceptron predictor.

Our study is directly related to the work by Quiñones et al. However, we identify that the association of the use of global branch-and-guard history with per-guard selective use of guard prediction can lead to the use of corrupted global branch-and-guard history (see Section 5.1). This further leads to significant performance loss when using state-of-the-art predictors such as TAGE [Seznec and Michaud 2006], GEHL [Seznec 2005], or hashed perceptron [Tarjan and Skadron 2005]. In Section 7, we illustrate that this phenomenon is much less marked on a perceptron predictor as used in Quiñones et al. [2007], but that due to the relative inefficiency of the perceptron predictor compared with state-of-the-art predictors, the overall performance remains relatively disappointing (see Section 7).

4. PREDICTING GUARDS ON THE ARM ISA

Most of the previously published studies on out-of-order execution of guarded instruction ISA target the Intel Itanium ISA [Intel Corp 2002]. For this ISA, the guard of a guarded instruction is the Boolean value contained in a guard register. In this study, we use the ARM-v7 ISA that features some different properties that we present next.

4.1. Guards on the ARM ISA

Instructions are guarded through a Boolean value computed from the value of four flags: the Negative flag (N), the Zero flag (Z), the Carry flag (C), and the Overflow flag (V). These flags are written by specific instructions, like compare instructions and specific arithmetic instructions [ARM 2014]. The values of the guards are not directly known through reading a register but require one to evaluate a logical formula on the

guard Mnemonics	Logical Formula
(EQ,NE)	$Z == 1$
(CS,CC)	$C == 1$
(MI,PL)	$N == 1$
(VS,VC)	$V == 1$
(HI,LS)	$(C == 1) \&\& (Z == 0)$
(GE,LT)	$N == V$
(GT,LE)	$(Z == 0) \&\& (N == V)$
AL	(always true)

Fig. 5. Possible guards on ARM v7.

Fig. 6. A guarded group is a group of instructions that use the same occurrence of a guard or its opposite. It ends when a flag-defining instruction is encountered.

flags. Guards are paired with a guard and its opposite. Figure 5 illustrates the set of possible guards. Pairs of opposite guards are grouped in the same entry, the logical formula corresponding to the first guard.

On ARM v7 ISA, conditional branches are guarded instructions.

4.2. Predicting Guarded Instructions

In many cases, several guarded instructions are using the same guard or the opposite guard: the original taken and not-taken paths. This leads to the concept of a guarded group of instructions, that is, the group of guarded instructions that use the same guard value or its opposite. A guarded group is associated with the use of the same occurrence of a specific guard. These instructions are not necessarily contiguous in the code since if-conversion leads to encapsulating and scheduling instructions from the taken path, instructions from the not-taken path, and instructions common to both paths in the same basic block. Figure 6 illustrates an example of two guarded groups. In our study, a guarded group starts at the first use of a guard and ends when a flag-defining instruction is encountered.

When guard prediction is used, one has only to predict the guard for the first instruction of the guarded group. In the remainder of the article, when referring to the global branch-and-guard history vector, we assume that the guard is appended only once to the history on its first encountering in the instruction flow, even when the same guard is used multiple times.

Upon the detection of a guard misprediction, the pipeline is flushed starting from the mispredicted instruction. Fetch is restarted at that instruction. One could be more selective and only flush instructions that are dependent on the misprediction. However, such a mechanism implies complex dependency tracking and thus has not been considered for this study.

5. BRANCH AND GUARD PREDICTION

5.1. Branch History Versus Branch-and-Guard History

The accuracy of a branch or a guard predictor depends on the prediction scheme, the predictor size, and the quality of the information that the predictor is exploiting. Global branch or path history is generally considered as the highest-quality information usable to predict branches. All recent branch predictor proposals such as TAGE [Seznec and Michaud 2006], GEHL [Seznec 2005], hashed perceptron [Tarjan and Skadron 2005], and SNAP [St. Amant et al. 2008] use global branch or path history as their main input vector. Adjunct predictors using other information inputs, such as the loop predictor [Gao and Zhou 2005; Seznec 2007] or a local history predictor component [Jiménez and Lin 2002; Seznec 2011a], may bring some extra but relatively marginal accuracy benefit.

Several studies [Tyson 1994; Quiñones et al. 2007] have pointed out that the global branch-and-guard history is often a better information vector than the global branch history, since branches are often correlated to some guards. Therefore, one would like to use the global branch-and-guard history predictor to predict both the branches and the guards for guarded instructions.

Branch and guard predictors are accessed at prediction time with a speculative history and updated at commit time with a nonspeculative history. The speculative global branch history used to read a prediction matches exactly the commit time global branch history on the right path. If all guards are predicted and the pipeline is flushed on every guard misprediction, then the same applies for the global branch-and-guard history.

However, systematically using guard prediction can lead to a performance loss compared to the use of split FPCM (see Section 7). Therefore, selective use of guard prediction as proposed in Quiñones et al. [2006] is appealing. For instance, one may only use guard prediction when the confidence of the prediction is high [Quiñones et al. 2006]. In that case, a low-confidence guard misprediction does not result in a pipeline flush, but it results in a corrupted speculative global branch-and-guard history. The branches and the guards predicted after the mispredicted guard are predicted using a wrong global branch-and-guard history.

On most predictors, a corrupted global branch-and-guard history induces reading wrong entries from the predictor, as illustrated in Figure 7. For example, on TAGE or GEHL, for the predictions just following the mispredicted guard, all the tables are read with a wrong entry number. The perceptron predictor is much less sensitive to this corruption, as illustrated in Figure 8. Since all predictor counters are accessed using only the program counter, only the counter associated with the mispredicted guard corrupts the prediction: if the predicted branch or guard is not strongly correlated with the mispredicted guard, then the absolute value of the counter will be small and the prediction result is likely to be unaffected.

5.2. The Branch-Only/Branch-and-Guard Predictor

To address the issue of corrupted branch-and-guard history mentioned earlier, we propose BO-BG (Figure 9), a hybrid predictor consisting of a global branch history component, BO, and a global branch-and-guard history component, BG. The predictor is used at fetch time to predict the branches and guards in the fetch group. A meta-predictor, META, **and** a Benefit-or-Loss heuristic hardware mechanism, BoL, are used to choose among the predictions flowing out from the two components.

The Benefit-or-Loss heuristic hardware mechanism, BoL, determines the execution mode: either *systematic guard prediction use* mode (SY-mode) or *high-confidence-only guard use* mode (HCO-mode). When running in SY-mode, all the guards are predicted

Fig. 7. Corrupted branch-and-guard history leads to reading a wrong predictor entry.

Fig. 8. Corrupted branch-and-guard history on a perceptron predictor often does not mean incorrect prediction.

and the predictions are systematically used; mispredictions are resolved in the execution stage and fetch is resumed at the first instruction of the guarded instruction group. That is, when executing on the correct path, the speculative branch-and-guard history used at prediction time is the correct branch-and-guard history.

Therefore, one can use the predictions (for branches and guards) that flow from both the BO and BG components; the META predictor is used to select between the two predictions.

On the other hand, when running in HCO-mode, all guards are predicted. But the prediction is used only when it is high confidence. The other guarded instructions are handled with the split FPCM mechanism. A low-confidence guard misprediction does not lead to a pipeline flush and therefore leads to the use of a corrupted speculative

Fig. 9. The BO-BG branch-and-guard predictor.

branch-and-guard history. Therefore, in HCO-mode, only the predictions flowing out from the BO component are used. The META predictor is not used in this case.

BoL is in charge of determining whether to run in SY-mode or HCO-mode. For this, we use a simple yet efficient heuristic. In both modes, the BO, BG, and META components are systematically updated at commit time as if the processor was running in SY-mode.

The META predictor is updated to reflect which component was providing the correct prediction. If both predictions were correct or both incorrect, the META predictor is not updated.

BoL uses simple signed 11-bit saturated counters that are updated according to Algorithm 1.

ALGORITHM 1: The Benefit-or-Loss heuristic

```

if Branch then
  if Pred(BO-BG)  $\neq$  Pred(BO) then
    BoL + = (Pred(BO-BG) correct) ? Penalty : -Penalty
  end if
end if

if Nonbranch guard then
  if Pred(BO) not high confidence then
    BoL + = (guarded group size)
    if Pred(BO-BG) incorrect then
      BoL - = Penalty
    end if
  else
    if Pred(BO-BG)  $\neq$  Pred(BO) then
      BoL + = (Pred(BO-BG) correct) ? Penalty : -Penalty
    end if
  end if
end if
  
```

The intuition behind the BoL heuristic is that (1) the performance benefit from a correct prediction of a guard is approximately proportional to the size of the guarded

group and (2) the performance loss (resp. benefit) from an extra misprediction (extra correct prediction, respectively) can be modeled by an average penalty.

Switching from HCO-mode to SY-mode implies restoring a correct speculative branch-and-guard history, that is, draining the complete pipeline. To avoid ping-ponging back and forth between HCO-mode and SY-mode, the HCO-mode (SY-mode, respectively) is triggered only when the BoL counter becomes lower than -512 (higher than 512, respectively).

In the remainder of the article, *Penalty* is an empirically determined constant. However, the best value for *Penalty* would depend on the precise core micro-architecture (issue width, pipeline depth, etc.). It can also dynamically depend on the application and on the application phase. Adaptive *Penalty* is left for future exploration.

In the remainder of the article, the BG and BO components of the BO-BG predictor will be TAGE predictors enhanced with a storage-free confidence mechanism close to the one described in Seznec [2011b]. For nonbranch guards and on a correct prediction provided by a counter whose value is 1, 2, -2, or -3, the prediction counter is incremented with probability $\frac{1}{32}$. This small modification avoids many high-confidence mispredictions without significantly modifying the global misprediction rate. For each of the TAGE components, 256Kbit storage budgets are considered, and a PC-indexed 1,024 five-bit entry META predictor is modeled.

Other global history predictors can also be considered, for example, GEHL [Seznec 2005], Hashed perception [Tarjan and Skadron 2005], or SNAP [St. Amant et al. 2008]. Global history perceptron predictors will also be considered in Section 7.1.2, since they present the particularity of being quite resilient to branch-and-guard history corruption. We do not consider any local branch (or guard history) component as their extra accuracy contribution is marginal, while their hardware implementation is quite tricky; in particular, predicting several branches and several guards per cycle with a local predictor and maintaining speculative local histories involve complex hardware logic.

6. EXPERIMENTAL FRAMEWORK

The experimental study for validating our propositions was built upon the Gem5 simulator [Binkert et al. 2011].

6.1. Simulator Parameters

The simulator models an aggressive four-way superscalar processor. Split FPCM is modeled for guarded instructions when guard prediction is not used. The processor also features a state-of-the-art conditional branch predictor, the TAGE predictor described in Seznec and Michaud [2006]. The store sets predictor [Chrysos and Emer 1998] is used to predict memory dependencies.

When the prediction is used, a guard misprediction is treated as for a branch misprediction: once it is detected after the execute stage, the overall pipeline is flushed and fetch is restarted on the correct path. For branch misprediction, fetch is restarted at the correct target of the branch. For guard misprediction, fetch is restarted on the mispredicted instruction.

The other characteristics are summarized in Figure 10.

The BASE configuration is the configuration **without** guard prediction and featuring only the BO branch predictor component.

We report simulation results (speedups over the BASE configuration) assuming a four-way superscalar processor, except in Section 7.8, which shows that trends are amplified for an eight-way superscalar processor.

OoO@1GHz	4-way	8-way
Memory	100 cycles 12.8GBps, Across a 128B bus	
Caches	L1D 4-way 64KB, 64B, 1 cycle L1I 4-way 64KB, 64B, 1 cycle L2 8-way 4MB, 64B, 8 cycles Stride Prefetcher for the L2	
TLBs	Perfect, 4K pages	
ROB	128 entries	256 entries
IQ	64 entries	128 entries
LSQ	128 entries (64L/64S)	256 entries (196L/64S)
Width (F/D/R/I/E/W/C)	4	8
Pipeline	12 stages	
FU(latency)		
IntAlu(1)	3	6
IntMultDiv(3/12*)	2	2
FpAlu(5)	2	4
FpMultDiv(4/9*)	2	4
Ld(2)	2	2
Str(1)	2	2
Intruction fetch	BTB 4-way, 1K entries RAS, 16 entries, WP corruption detection 2 x TAGE 1+12 components, 15 Kentries META, 1 Kentries, 5-bit Perceptron, 40-bit history, 1 Kentries	
Misprediction penalty	15 cycles	15 cycles

Fig. 10. Simulator configuration overview. *not pipelined.

Benchmarks	inputs	IPC (BASE)		% guarded	
		4-way	8-way	with branches	without branches
400.perlbench	checkspam diffmail	1.47	1.73	16.59	4.63
401.bzip2	chicken combined liberty program source text	2.21	2.7	16.46	4.68
403.gcc	c-typeck 166 cp-decl expr scilab 200	1.67	2.15	36.11	24.47
416.gamess	cytosine h2ocu2+	3.17	4.84	6.7	2.89
429.mcf	ref	0.71	0.81	24.5	6.1
435.gromacs	ref	3.01	4.8	4.62	1.02
436.cactusADM	ref	3.53	4.3	0.09	0.02
444.namd	ref	2.62	3.83	8.54	5.18
445.gobmk	13x13 nngs trevorc trevord	1.8	2.29	18.05	6.86
453.povray	ref	1.81	2.35	8.09	1.9
456.hmmmer	nph3 retro	3.25	4.87	17.41	14.67
458.sjeng	ref	1.82	2.3	18.76	7.05
459.GemsFDTD	ref	2.53	3.89	1.06	0.001
462.libquantum	ref	1.86	2.11	23.56	13.42
464.h264ref	sss baseline main	2.57	3.21	6.4	2.41
470.lbm	ref	2	2.34	0.6	0.02
471.omnetpp	ref	0.81	0.91	17.1	4.33
473.astar	BigLakes rivers	1.4	1.76	15.62	3.59
483.xalancbmk	ref	1.83	2.45	21.09	3.4

Fig. 11. Benchmarks, their inputs, their IPC for the BASE four-way and eight-way configurations, and the ratio of guarded instructions over the total number of instructions.

6.2. Benchmarks

The simulated benchmarks constitute a subset of the Spec 2006 benchmarks set [SPEC 2006] listed in Figure 11. To reduce the amount of simulation time, we use the Simpoint methodology [Hamerly et al. 2005] to summarize each benchmark in a set of 100 million instructions slices. Each slice is representative of a part of the benchmark execution

Fig. 12. Systematic guard prediction use: TAGE-based predictor.

and is affected by a weight representing the portion that it represents in the execution. For each benchmark, the illustrated results are the weighted mean of simulations on the set of slices [Hamerly et al. 2005]. Figure 11 displays the weighted mean of the Instruction Per Cycle (IPC) count for each benchmark for the four-way and eight-way BASE configurations.

As we target the ARM instruction set, some of the benchmarks or some of their input sets are missing. There are three reasons that some benchmarks are missing: (1) the binary produced by our cross-compiler is not executable on a native ARM architecture; (2) the binary is not executable on *qemu-arm* [Bellard 2012], which was used to compute the basic block vector (BBV) needed to compute the simpoins; and (3) the Gem5 ARM-v7 simulator is not able to run them. In the end, we were able to run 12 integer benchmarks (the complete set of integer benchmarks) and seven floating-point benchmarks. Some benchmarks are used with several inputs (all the inputs that are working are used). In total, we were able to simulate 38 different workloads. For each benchmark, the results shown are the average results of its different inputs.

The binaries were generated with the gcc compiler using the O3 optimization level. The gcc decision to if-convert a branch mainly depends on the number of instructions that are controlled by the branch. By default, for the ARM target, this number is set to 4.

6.3. Ratio of Guarded Instructions

Figure 11 also lists the ratio of guarded instructions per benchmark. The first column presents the total percentage of guarded instructions. This includes the conditional branches. The second column excludes conditional branches.

For all benchmarks, conditional branch instructions represent a large part of the guarded instructions. However, some benchmarks like *401.bzip2*, *403.gcc*, *445.gobmk*, and *456.hmmer* contain a significant portion of effective guarded instructions. Some other benchmarks, like *436.cactusADM*, *459.GemsFDTD*, and *470.lbm*, feature nearly no effective guarded instructions.

A simple optimization to save energy would be to monitor at runtime the ratio of effective guarded instructions and to turn off the guard prediction when this ratio is under a predefined threshold.

7. EXPERIMENTAL RESULTS

7.1. Branch History Versus Branch-and-Guard History

Unless explicitly mentioned, simulation results are reported in relative speedups over the BASE four-way configuration.

7.1.1. Systematic Guard Prediction Use. Figure 12 reports simulation results assuming that guard predictions are systematically used for three predictors: a BO-history TAGE, a BG-history TAGE, and the BO-BG predictor without using the BoL heuristic.

Fig. 13. High-confidence-only guard prediction use: TAGE-based predictor.

Fig. 14. High-confidence-only guard prediction use: perceptron-based predictor.

As expected, systematic guard prediction is effective at enabling performance gain on most applications. But on some applications (e.g., *444.namd*, *456.hmmmer*, *458.sjeng*, *464.h264ref*), performance losses are encountered. The performance loss is most dramatic on *456.hmmmer*. Therefore, systematic guard prediction use should not be considered for implementation in real hardware. One can also remark that using branch-and-guard history is often beneficial (e.g., on *401.bzip2*, *416.games*, or *462.libquantum*), but not systematically (e.g., on *429.mcf* or *471.omnetpp*). As expected, the BO-BG predictor slightly outperforms its two components.

7.1.2. High-Confidence-Only Guard Prediction Use. Figure 13 illustrates simulation results assuming that guard predictions are used only when there is high confidence. As anticipated, the BG-history TAGE results most often in lower performance than the BO-history TAGE, because the speculative branch-and-guard history is corrupted by incorrect guard predictions. Restricting the prediction usage to high-confidence branches appears as an effective filter to eliminate performance loss due to guard mispredictions for the BO predictor. The BO-history TAGE should even be considered as a valid design point for effective designs, since it outperforms the BASE design at the exception of a marginal loss on *470.lbm*.

The BO-BG predictor (not illustrated) is not worthy as a design point since its BG component has poor behavior.

7.2. Perceptron Predictor

We ran similar simulations assuming perceptron predictors as components instead of TAGE. We assumed a 40-bit history length and 1K entry predictors, that is, a 41Kbyte perceptron predictor. As mentioned in Section 5.2 and according to Quinones et al. [2007], the perceptron predictor should be much more resilient to branch-and-guard history corruption than TAGE.

Figure 14 reports results for this experiment. Prediction confidence is estimated as follows. An extra counter is added to each perceptron entry to monitor the correctness of the predictions. This confidence counter is incremented on a correct prediction and reset on a misprediction. High confidence is considered on saturated counters only. As the perceptron predictor is not our main target, we run multiple simulations varying

Fig. 15. Impact of the *Penalty* constant in BoL heuristic.

Fig. 16. Use of predicted guarded instructions.

the counter width from 0 to 7 bits, and we only illustrate the best configuration for each benchmark. The reported results should therefore be considered as an upper limit.

First, the perceptron predictor alone without guard prediction is quasi-systematically outperformed by the TAGE predictor, and often by a quite significant margin. As reported by Quiñones et al. [2007], branch-and-guard history associated with high-confidence-only guard use allows one to systematically outperform the perceptron predictor without suffering a major performance loss on any benchmark. However, the benefit is limited and the performance is lower than BO-history TAGE + high-confidence-only guard prediction use. The performance often does not reach the level of our BASE using a TAGE predictor without any guard prediction use.

7.3. BO-BG Predictor with BoL Heuristic

Figure 15 reports simulation results assuming the BO-BG TAGE predictor assuming respectively 32, 64, and 128 as the *Penalty* constant in the BoL heuristic. As expected, BO-BG/BoL reaches higher performance than just running in SY-mode or just running in HCO-mode. It also slightly outperforms the best of the two modes for all benchmarks.

However, the performance impact of the *Penalty* constant is relatively low. This tends to indicate that in most code sections, one of the two modes, SY-mode or HCO-mode, has a clear performance benefit over the other mode.

In the remainder of the article, we will assume that *Penalty* = 64.

7.4. Performance Analysis

The performance benefit allowed by our proposal comes from three different factors. First, guard prediction eliminates the execution of the extra instruction in split FPCM and eliminates the execution of the predicted guarded instructions whose guard is predicted false. Second, it greatly simplifies the dependency chain, eliminating the artificial data dependency created by guarded execution and for predicted true guarded instructions breaking the dependency chain with the guard writer.

Figure 16 illustrates the ratio of predicted guarded instructions used in the different benchmarks. On many applications, the most significant part of the application runs in SY-mode. Even on applications running essentially in HCO-mode, a very

Fig. 17. Branch misprediction rates.

Fig. 18. Instruction queue size impact.

Fig. 19. Relative guard prediction benefit for different instruction queue sizes.

significant part of the guards is predicted with high confidence with a minimum of 60% on *456.hmmer*.

It should be noted that the benchmarks experiencing slowdown when SY-mode is enforced (Figure 12) are those that exhibit the lowest ratio of high-confidence guard predictions, as, for example, *456.hmmer*, *458.sjeng*, or *462.libquantum*.

Second, the overall conditional branch misprediction rate is sometimes significantly reduced through use of a hybrid predictor using both branch history and branch-and-guard history as illustrated in Figure 17, for example, on *401.bz2* or *473.astar*.

7.5. Reducing the Instruction Queue Pressure

By using guard prediction, the number of instructions that enter the instruction queue is substantially reduced. First, for predicted false guard instructions, only the first instruction in the guarded group enters the instruction queue. Second, the predicted guarded instructions are not split.

Figure 18 illustrates the performance assuming a 32-entry instruction queue instead of a 64-entry instruction queue. Without guard prediction, the impact of instruction queue size reduction is important on a few benchmarks (e.g., *444.namd*, *456.hmmer*, and *473.astar*). Figure 19 illustrates that the relative benefit of using guard prediction is generally higher when the instruction queue size is smaller (e.g., *403.gcc* and *456.hmmer*).

Fig. 20. Split FPCM versus direct FPCM.

Fig. 21. Double-size TAGE versus BO-BG/BoL64.

7.6. Split FPCM Versus FPCM

Up to now, we have assumed that split FPCM execution is used for guarded instructions. Direct implementation of FPCM would save an operation per guarded instruction (if guard prediction is not used) but would require an extra register operand per guarded instruction (see Section 2.3). Since in most cases, guarded instructions are predicted, the potential benefit of a direct implementation of FPCM can be much lower when guard prediction is used.

Figure 20 illustrates the performance benefit of using FPCM instead of split FPCM. There is a small performance benefit when guard prediction is not used, less than 3%, except on *403.gcc*, up to 10%. This benefit vanishes when guard prediction is used through our BO-BG/BoL proposal. This is expected since for most benchmarks, most of the guarded instructions are predicted.

Therefore, the extra hardware complexity associated with the direct implementation of FPCM instead of split FPCM is not worth paying for when guard prediction is used through our BO-BG/BoL predictor.

7.7. Hardware Efficiency

Figure 21 compares a double-size TAGE predictor against the BO-BG/BoL predictor. Except for *444.namd*, *445.gobmk*, and *470.lbm*, the BO-BG/BoL predictor largely outperforms the double-size TAGE predictor. In practice, only *445.gobmk* shows significant performance improvement when the size of the TAGE predictor is doubled.

Thus, it seems that adding a guard predictor to a branch predictor is a better way to spend an increased budget than just increase the size of the branch predictor itself.

7.8. Wide-Issue Superscalar Processor

The benefit of our BO-BG/BoL proposal is growing, especially when one considers a more aggressive implementation featuring a wide-issue out-of-order engine. Figure 22 illustrates this on an eight-way superscalar processor.

As for the four-way configuration, using 64 as the BoL *Penalty* constant appears as a good tradeoff. In practice, except for *401.bzip2*, *444.namd*, *445.gobmk*, and *456.hmmer*, the value of the *Penalty* constant has very little impact on the performance results.

Fig. 22. Guard prediction on an eight-way issue superscalar processor. BASE is eight-way.

The speedup over an eight-way base superscalar grows to up to 18% on *403.gcc* and the relative speedup is most often higher for the eight-way issue than for the four-way issue.

8. CONCLUSION

ARM-based processors are becoming ubiquitous in many modern appliances including smartphones and tablets. The demand for high performance pushes manufacturers of ARM processors to use the same techniques that have been used for the past two decades on PCs and server processors, including wide-issue superscalar processors. Despite the introduction of 64-bit ARM-v8 ISA, the demand for a lower-cost ARM v7 processor will continue to be dominant for many years, due to cost and power issues. The 32-bit ARM v7 ISA features guarded instructions. Providing an efficient solution to efficiently execute guarded instructions out of order is challenging due to the multiple definition problem.

Fortunately, predicting the instruction predicates addresses the multiple definition problem. In this article, we have shown that a state-of-the-art global branch history predictor can be adapted to predict both branches and guards. We have proposed BO-BG, a hybrid predictor combining a branch-only history component and a branch-and-guard component. Unfortunately, systematic use of guard prediction and sometimes poor overall guard prediction accuracy lead to poor overall performance, sometimes significantly worse than the performance without guard prediction.

As such, we have proposed BoL, a simple heuristic that evaluates dynamically the potential gain associated with systematic use of guard prediction. BoL is used to control two execution modes: systematic guard prediction use, or SY-mode, and high-confidence-only guard prediction use, or HCO-mode.

Our experiments show that the association of BO-BG with BoL allows one to achieve high out-of-order execution performance on a guarded instruction set. The processor runs in SY-mode on code sections where the guards are highly predictable and/or branch-and-guard history allows one to reduce the conditional misprediction rate. The processor runs in HCO-mode on regions where the guard misprediction rate is high, but still a significant number of guards are predicted since their prediction is high confidence. Therefore, the simple but relatively inefficient hardware associated with split false predicated conditional move is sufficient for executing the few nonpredicted guarded instructions. Significant performance benefits are encountered on most applications, while applications with poorly predictable guards do not suffer significant performance loss. This benefit will be even higher in future very-wide-issue superscalar processors.

Therefore, guarded ISA is not an obstacle anymore for the implementation of efficient out-of-order execution. BO-BG/BoL could be considered as an opportunity to allow more aggressive use of guarded instructions by the compiler. For instance, on applications featuring a few suspected poorly predictable branch statements inside

loops (e.g., *456.hmm*), the branch control-flow instructions could be handled through if-conversion. At runtime, depending on the global guard predictability, the hardware will decide whether to execute them either in HCO-mode or in SY-mode.

REFERENCES

- J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 177–189.
- Alpha. 1999. Alpha 21264 Microprocessor Hardware Reference Manual. Compaq Computer Corporation.
- ARM. 2014. ARM Architecture Reference Manual. ARM v7-A and ARM v7-R edition.
- Fabrice Bellard. 2012. QEMU. Retrieved from http://wiki.qemu.org/Main_Page.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.
- Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. 1996. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. *Int. J. Parallel Program.* 24, 3 (1996), 209–234.
- George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. In *ISCA'98: Proceedings of the 25th Annual International Symposium on Computer Architecture*. 142–153.
- Weihaw Chuang and Brad Calder. 2003. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th Annual International Conference on Supercomputing*. 183–192.
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490.
- Hongliang Gao and Huiyang Zhou. 2005. Adaptive information processing: An effective way to improve perceptron predictors. *J. Instruction-Level Parallelism* 7 (2005).
- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. SimPoint 3.0: Faster and more flexible program phase analysis. *J. Instruction Level Parallelism* 7 (Sept. 2005).
- Intel Corp. 2002. Intel Itanium Architecture Software Developers Manual. Volume 3: Instruction Set Reference. (2002).
- Daniel A. Jiménez and Calvin Lin. 2002. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.* 20, 4 (2002), 369–397.
- Hyesoon Kim, Jose A. Joao, Onur Mutlu, and Yale N. Patt. 2006. Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 53–64.
- Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. 2005. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. 43–54.
- Dimitris Pnevmatikatos and Guri Sohi. 1994. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. 120–129.
- Eduardo Quiñones, Joan-Manuel Parcerisa, and Antonio Gonzaleiz. 2007. Improving branch prediction and predicated execution in out-of-order processors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, Washington, DC, 75–84. DOI: <http://dx.doi.org/10.1109/HPCA.2007.346186>
- Eduardo Quiñones, Joan-Manuel Parcerisa, and Antonio Gonzalez. 2006. Selective predicate prediction for out-of-order processors. In *Proceedings of the 20th Annual International Conference on Supercomputing*. 46–54.
- André Seznec. 2005. Analysis of the O-GEometric history length branch predictor. In *ISCA*. 394–405.
- André Seznec. 2007. The L-TAGE branch predictor. In *J. Instruction Level Parallelism*.
- André Seznec. 2011a. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Conference on Microarchitecture*. 117–127.
- André Seznec. 2011b. Storage free confidence estimation for the TAGE branch predictor. In *Proceedings of the 17th IEEE International Conference on High Performance Computer Architecture*. 443–454.

- Andr Seznec and Pierre Michaud. 2006. A case for (partially) tagged Geometric History Length Branch Prediction. *J. Instruction Level Parallelism* (Feb. 2006).
- Beth Simon, Brad Calder, and Jeanne Ferrante. 2003. Incorporating predicate information into branch predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 53–64.
- James E. Smith. 1981. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*. 135–148.
- SPEC. 2006. SPEC CPU2006. Retrieved from <http://www.spec.org/cpu2006/>.
- Renee St. Amant, Daniel A. Jimenez, and Doug Burger. 2008. Low-power, high-performance analog neural branch prediction. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, 447–458. DOI : <http://dx.doi.org/10.1109/MICRO.2008.4771812>
- David Tarjan and Kevin Skadron. 2005. Merging path and gshare indexing in perceptron branch prediction. *TACO* 2, 3 (2005), 280–300.
- Gary Scott Tyson. 1994. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. 196–206.
- Perry H. Wang, Hong Wang, Ralph-Michael Kling, Kalpana Ramakrishnan, and John Paul Shen. 2001. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. 15–25.

Received May 2014; revised August 2014; accepted October 2014