A Hardware-Based Method for Dynamically Detecting Instruction-Isomorphism and its Application to Branch Prediction

Kypros Constantinides*
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
USA
kypros@umich.edu

Yiannakis Sazeides
Department of Computer Science
University of Cyprus
Cyprus
yanos@cs.ucy.ac.cy

Abstract

This paper proposes a hardware-based heuristic method for implementing various transformations and detecting isomorphism in the dynamic dependence graph of a program. This enables on the fly identification of isomorphic instructions which may be useful for improving the performance of several microarchitectural mechanisms. This work considers the application of the proposed method to conditional branch prediction. The empirical results using SPEC benchmarks suggest that the proposed method may be useful for increasing prediction accuracy and improving performance. Specifically, is shown for a 4-way processor that a 16KB gshare predictor combined with a 16KB overriding isomorphic predictor can achieve better performance than either a 32KB gshare or a 32KB combining gshare/bimodal predictor.

1 Introduction

The pursuit for higher computing performance has lead to the development of several basic microarchitectural mechanisms - pipelining, caches, predictors etc - that can be found in virtually all high-performance microprocessors. Most of these mechanisms exploit one or combination of dynamic program properties, such as locality and predictability. One other recently introduced dynamic program property, that may be useful for improving microarchitectural performance, is instruction-isomorphism [11].

A dynamic instruction instance is said to be isomorphic if its component-graph - information derived from the instruction and the dynamic dependence graph of a program - is identical to the component-graph of an instruction executed earlier. Instruction-isomorphism was shown [11] to occur frequently during program execution only when a number of transformations are applied on the dynamic data dependence graph to remove "useless" computation.

The presence of instruction-isomorphism suggests repetition in the transformed program structure. Instruction- isomorphism, therefore, builds on earlier instruction-repetition and instruction-reuse work that demonstrated instructions to often repeat with same input-output values [14], and to repeat, less often, with the same dependences [13]. Instruction-isomorphism is different from instruction-reuse though, because it extends the scope of the dependence graph that can be considered and employs transformations on the dependence graph to facilitate isomorphism.

The previous instruction-isomorphism work [11] assumed, however, a perfect implementation of the various graph transformations and a perfect detection of graph isomorphism. This paper proposes a heuristic hardware-based method for implementing various transformations and detecting on the fly isomorphism in the dynamic dependence graph.

The notion of instruction-isomorphism may provide a new perspective on several microarchitectural issues and may eventually lead to better performing or even new microarchitectural mechanisms. Potential applications of the proposed instruction-isomorphism detection mechanism include various types of predictors such as branch, value and dependence predictors [12, 6, 9].

This work considers the application of the proposed isomorphism detection method, called *IDM*, to conditional branch prediction. An empirical analysis for an out-of-order processor using SPEC benchmarks suggests that an IDM based predictor may be useful for increasing the performance of a 4-way processor - often by more than 4% - as compared to other predictors with similar cost.

2 Isomorphism Basics

This section reviews instruction-isomorphism related definitions and also describes theoretical transformations proposed to uncover isomorphism. Most of the information in this section comes from [11].

2.1 Isomorphic-Equality, Pseudo-Isomorphism and Non-Isomorphism

A dynamic instruction instance is said to be isomorphic if its **component-graph** is identical to the component-graph of a dynamic instruction instance executed earlier. The component-graph of a dynamic instruction instance can include information from the instruction, its dynamic data dependence graph, and its input data (values read but not produced by program instructions). Fig. 1 shows an example dynamic instruction sequence (Fig. 1.a) and the component-graph for the branch instruction $beq \$4, \$5, 25$ (Fig. 1.b).

During program execution each dynamic instruction will either be **isomorphic** to one or more previously executed instructions or **non-isomorphic** to any previously executed instruction.

*This work was done while the author was with the University of Cyprus

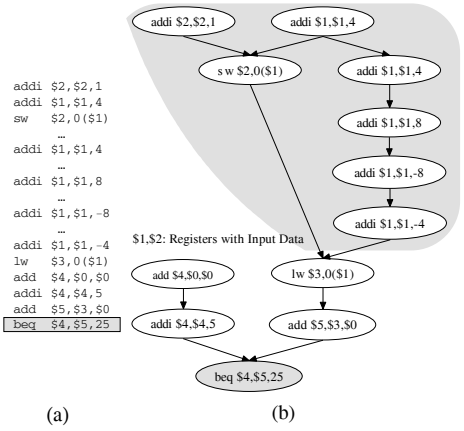

Figure 1. (a) Example Dynamic Sequence, and (b) the Component-Graph for a Branch Instruction

When two instructions are isomorphic to each other and their outputs are equal we have a case of **isomorphic-equality**, and when their outputs are not equal a case of **pseudo-isomorphism**.

In theory, when instruction component-graphs include all the information that affects them, pseudo-isomorphism should not occur because two isomorphic instructions will represent exactly the same computation. However, real life constraints can lead to pseudo-isomorphism. Two of the main causes of pseudo-isomorphism are imperfect encoding of dependence graphs, and the application of **unsafe memory transformations** to represent dependences through memory. Unsafe memory transformations may be essential because detecting on the fly exact read-after-write and read-after-read memory dependences is particularly difficult [8]. Therefore, if the component-graph of the example branch instruction was (unsafely) transformed to not include the subgraph for the loaded data (shaded part of Fig. 1.b), the branch could have been pseudo-isomorphic to another branch that read a different value from memory.

The above indicate that during program execution the frequency of the three types of isomorphic behavior: non-isomorphism, isomorphic-equality and pseudo-isomorphism, depends on the information included in the component-graphs of instructions. Next we describe some transformations that can change the structure and content of a component-graph and possibly its isomorphic behavior.

We note, that is possible to have component-graphs that include both control and data dependences. However, this paper considers component-graphs with only data dependences. Control dependences are not considered because once a component-graph is formed control dependences do not influence the dataflow in the component-graph.

2.2 Component-Graph Transformations

This section describes component-graph transformations that may convert non-isomorphism to isomorphic-equality. The basic idea behind most transformations is to remove information from the component-graph of an instruction that does not affect its outcome, such as data movement through registers or memory.

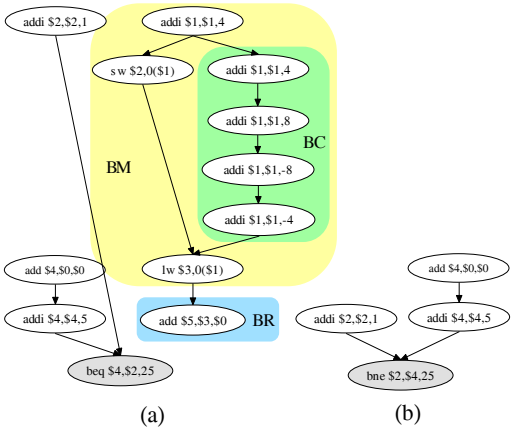

Figure 2. Component-Graph Transformations

These transformations may lead to smaller component-graphs that are more likely to be isomorphic-equal.

Bypass-Computation (BC): The purpose of this transformation is to bypass two dependent instructions when the one adds(subtracts) an immediate value that the other subtracts(adds) the same immediate value. This transformation can be applied to more than two instructions at a time and is mainly aimed to reduce the number of unique components defining the stack pointer (SP), which typically form a very long dependence chain.

Bypass-Memory Move (BM): bypass memory communication instructions (store and loads) by having the consumer of a load instruction linked directly to the instruction that produced the value that was stored in memory. The idea of memory bypassing was introduced in [9] and here is leveraged to transform component-graphs and facilitate isomorphism.

Bypass-Register Move (BR): bypass instructions that move data between two registers. This is mainly useful to eliminate the overhead computation due to call/return convention.

Conditional-Branch-Reversal (CBR): the goal of this transformation is to allow for branches that test for complementary conditions to be isomorphic. For example, a branch that tests for equality can be isomorphic to a conditional branch that tests for inequality. When this occurs the output of the former branch will be the complement of the latter. The following pairs of conditional branches are allowed to be isomorphic (these are specific to the instruction set used in this study [2]): *bne-beq, blez-bgtz, bltz-bgez* and *bc1f-bc1t*.

Commute (COM): this transformation allows two instructions with the same commutative optype to be isomorphic irrespective of the ordering of their input operands.

The workings of the above transformations are illustrated in Fig. 2. Fig. 2.a shows what information is removed when applying the BC, BM and BR transformations on the component-graph of Fig. 1.b. The end-result is a smaller component graph with the beq instruction directly dependent on the $addi\$2, \$2, 1$. Furthermore, by applying the CBR and COM transformations, the component-graph in Fig. 2.a becomes isomorphic with the component-graph in Fig. 2.b.

Previous work has found instruction-isomorphism to occur frequently when isomorphism-detection and the above component-graph transformations are implemented perfectly and off-line. The next section proposes a practical method

for implementing the transformations and detecting instruction-isomorphism on the fly.

3 IDM: A Hardware-Based Method for Implementing Component-Graph Transformations and Detecting Isomorphism

This section describes IDM: a hardware-based method that implements various component-graph transformations and can detect instruction-isomorphism dynamically. Although most of the basic functionality of IDM will be common across all its applications, its implementation may vary depending on the type of architectural information that is available at the stage of the pipeline the mechanism is used. In this paper we describe an IDM placed at the front-end of a processor.

3.1 IDM's Logical Organization

The mechanism consists of four units: the register-signature-file (RSF), the component-graph encoding/transformation mechanism (CGET), the Memory Signature File (MSF), and the instruction-isomorphism detection table (IDT). The IDM is shown pictorially in Fig. 3.

The RSF is accessed with the source architectural register names of an instruction to read the **signatures** - encoded component-graphs - stored in the RSF by the most recent instruction(s) that had these registers as a destination.

The CGET takes the source signatures of an instruction, information from the current instruction (such as the optype) and creates a new signature. This new signature represents the encoded/transformed component-graph of the instruction. If the instruction writes to a register the new signature is written to the destination register in the RSF. This procedure can encode in a signature information from all instructions that the proposed method can trace directly or indirectly a data dependence.

To determine if an instruction is isomorphic to a previously executed instruction, its signature, produced by CGET, is used to access the IDT. The IDT returns whether an instruction is isomorphic and some information about this component-graph past behavior. The specific information returned by the IDT depends on the IDM's application. Specific details about an IDM are presented in Section 4 where we consider its application to branch prediction.

The MSF is a signature cache that contains in each entry an address signature and a data signature. The purpose of the MSF is to implement the memory bypassing transformation (Section 2). To accomplish this, dependences between memory instructions are established using address signatures, and then data signatures are propagated between instructions with matching address signatures. Due to space limitation we do not consider further in this report the use of the MSF to perform memory bypassing.

For the remaining paper, component-graphs cannot go past memory dependences, and load instructions update the RSF using a signature that better represents their missing component-graph. We refer to this as **unsafe memory bypassing (UMB)** which is discussed next.

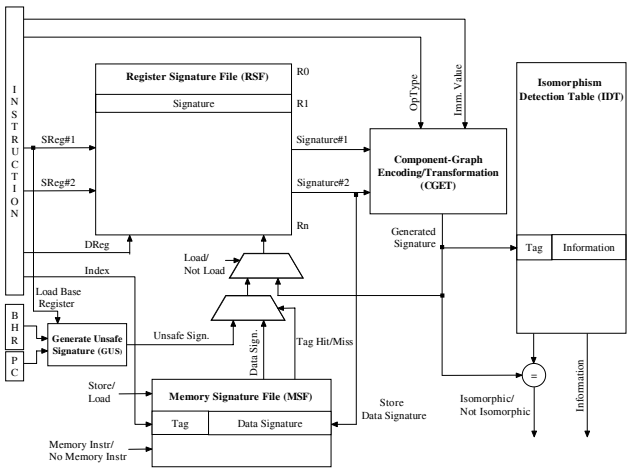

Figure 3. IDM's Logical Organization

3.2 Unsafe Memory Bypassing (UMB)

Unsafe memory transformations are essential because dynamically tracking all read-after-write and read-after-read memory dependences is particularly difficult. As shown in previous work [8], there can be numerous memory dependences, can be very far apart in terms of dynamic instructions, and when needed speculative are not always regular to be learned using known history based prediction schemes. Several options exist for implementing the unsafe memory bypassing by updating the destination register of a load as follows: with a fixed code indicating unsafe-memory transformation, with the PC (program counter) of the load, with the PC of the load xored with the BHR (branch history register), and with the load's address signature.

We have established empirically that for the application of IDM to branch prediction (Section 4) the best choice for an unsafe signature is the *xor* of the BHR with the PC of the load instruction. It was also found experimentally that for better performance the number of history bits used should not be the same for each load. This may be the case because some loads always produce the same output value and their PC is sufficient to identify their behavior, whereas for some other loads branch history bits may be useful to capture correlating behavior. A simple classification scheme that performed satisfactorily was to decide the number of history bits used based on the memory segment a load accesses. The memory segment to be accessed can be approximated using the load's base register: a gp (global pointer) base register is assumed to indicate accesses to the data segment, an sp (stack pointer) base register is assumed to indicate accesses to the stack segment, and all $other$ base registers to represent accesses to the heap segment. The analysis showed consistently that loads reading the data segment (with base register gp) required fewer BHR bits than loads from the stack or the heap. The above experimental analysis is not reported due to limited space.

Henceforth for UMB the PCBHR transformation is assumed. In Fig. 3, the **generate-unsafe-signature (GUS)** unit produces unsafe signatures based on the PC, the BHR and the base register of the load instruction.

A consequence of unsafe transformations is that there may be a need to distinguish component-graphs with and without

unsafe transformations. One way to determine the safeness of a signature is by adding an additional field in the signature, the **safe-bit**. This bit is simply propagated by all instructions and is set to unsafe only by loads that are unsafely transformed. For a two input instruction the safe bit is computed as a logical *or* of its inputs safe bits. The safe-bit may be useful for assigning higher degree of confidence to safe signatures or for guiding replacement in IDT.

3.3 Implementing the Component-Graph Encoding and Transformation Engine (CGET)

Component-graphs are represented in encoded-form using signatures. A signature consists of several fields aimed to uniquely identify a component-graph:
OptypeSign encodes the operation-types of the instructions in a component graph,
DataSign encodes all the immediate values of the instructions in a component-graph,
Depth contains the length of the longest dependence chain, and **Safe Bit** indicates whether the signature includes any unsafe memory transformation (Section 3.2).

A signature for a dynamic instruction is encoded on the fly by the CGET using the instruction's source registers signatures, information about the instruction, and current processor state. We considered four cases of encoding that depend on an instruction's source registers and optype: (a) two source registers and commutative operation (add, and, beq etc), (b) two source registers but not-commutative operation (sub, slt, blt), (c) one source register (addi, andi, slti etc), and (d) no register input (load immediate).

For a two register input and commutative instruction, the new OptypeSign is computed as the *xor* of the two source OptypeSigns and the operation-type of the current instruction, followed by a 1-bit left-rotation. The same function is used for computing the new DataSign. The depth is computed by adding a 1 to the maximum value of the source depths. This encoding procedure, shown in Fig. 4.a, implements the Commute (COM) transformation (Section 2) by not distinguishing the source operands ordering.

For a two register input non-commutative instruction the encoding is the same as for a commutative instruction with the difference that the "left" OptypeSign and DataSign are left-rotated by 1 bit before they are used to compute the new codes. This encoding aims to distinguish two non-commutative instructions that have the same input signatures but in opposite order.

The encoding for instructions with one input register and no input register are similar in spirit with the above. Fig. 4.b shows how a one input instruction is encoded.

The abovementioned method attempts to provide fast, concise and accurate representation of component-graphs. But this is done in a heuristic manner and therefore is possible for a non-isomorphic instruction to appear pseudo-isomorphic. The rotation applied before updating the OptypeSign and DataSign fields attempts to reduce signature pseudo-isomorphism. The importance of this encoding step, referred to as **ROT**, will be demonstrated experimentally.

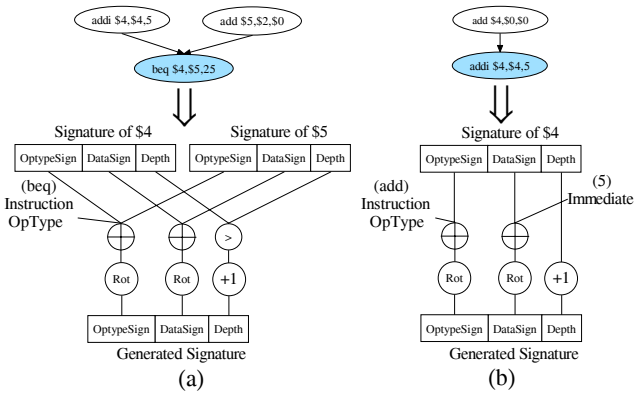

Figure 4. The Encoding Procedure (a) Commutative Instruction, and (b) One input instruction

3.4 Implementing Component Graph Transformations

Section 2 introduced several component-graph transformations. We have already described how to implement two of them, bypass-memory move (BM) and commute (COM), below we describe how to implement the rest in the CGET.
Bypass-Computation (BC) This transformation is applied to add instructions that have an immediate operand. For these instructions the encoding for the Optype and DataSign fields is slightly modified depending on the sign of the immediate value. Specifically, if the immediate value is positive the encoding is performed as in Fig. 4.b, otherwise if it is negative the two fields are encoded in "reverse": first they are rotated, in the opposite direction of Fig. 4.b, to the **right** and then the OpType signature is xored with the new optype where as the DataSign is xored with the magnitude of the immediate value. This encoding enables to remove from component-graphs the Optype and Immediate values of consecutive instructions that add(subtract) and then subtract(add) the same immediate value.
Bypass-Register Move (BR) This transformation is implemented in CGET by simply propagating the source register signature to the destination register when: (a) the optype of an instruction is a move, or (b) one of the sources is the zero register and the optype of the instruction ensures the output will be the same as the contents of the other register (for example add, xor, or).
Conditional-Branch-Reversal (CBR) This is realized directly in the encoding process by using always one opcode for pairs of conditional branch types that are allowed to be isomorphic (Section 2).

3.5 Pipelining Issues

The IDM can be placed in a processor pipeline after the stage where decoded fetched instructions are available, since the encoding/transformation engine requires information such as the optype, immediate value, source registers and destination register.

The register signature file, RSF, can be accessed using archi-

tectural or physical register names and most likely will need to be updated speculatively. For performance reasons the RSF may need to be recovered in case of misspeculation and therefore will need to be checkpointed [1]. One way to reduce checkpoint memory cost is to not have an entry in the RSF for each register. This may be acceptable when there is no benefit by maintaining all register signatures.

If high speed isomorphism-detection is required the various IDM structures may need to be multiported and the encoding mechanism will need to perform, in parallel, encoding of several instructions that are possibly dependent. Encoding in parallel signatures of dependent instruction can be performed quickly because the encoding for all instructions types requires simple operations such as the bitwise *xor*. A detailed implementation of parallel signature encoding is not presented due to limited space.

3.6 Applications of the Dynamic Isomorphism Detection Method

Isomorphism detection can be applied to improve the performance of several types of predictors. Branch, and value predictors may be improved by using the proposed isomorphism-detector to learn the value that is associated with the signature of a component-graph and when the same signature repeats to predict the previous value. To accomplish this the IDT will need to store in its entries the value associated with a given signature.

One other application of the isomorphism-detector is to assist the storageless value predictor [18] for deciding whether the signature of an instruction is isomorphic to the signature in a register and using this information to decide how to register-rename instructions.

Memory-dependence prediction and other memory related optimizations [8] can also benefit from isomorphism detection. The MSF can be used to determine when two address signatures are isomorphic and therefore dependent. This can help with the scheduling of memory instructions or help bypass memory communication through register renaming [8].

We believe that an isomorphic based predictor is probably better to be used as an additional component rather as stand-alone to capture only those predictions that other more cost-effective schemes can not predict correctly.

Next we describe the application of the proposed method for implementing an isomorphic-based branch predictor.

4 Isomorphic-Based Branch Predictor

To exploit the isomorphism exhibited by dynamic conditional branches an overriding prediction approach is proposed. This is shown in Fig. 5, where an example pipeline combines a fast base predictor (this can be any fast predictor proposed to date) with a slower isomorphic based predictor. Isomorphism-detection needs to wait for decoded instruction information and as a result the isomorphic branch prediction will come few cycles after the base prediction is available but many cycles before the actual execution of a branch. Consequently, the isomorphic prediction will be used to validate and possibly override the prediction provided by the base predictor. A confidence-estimator is employed

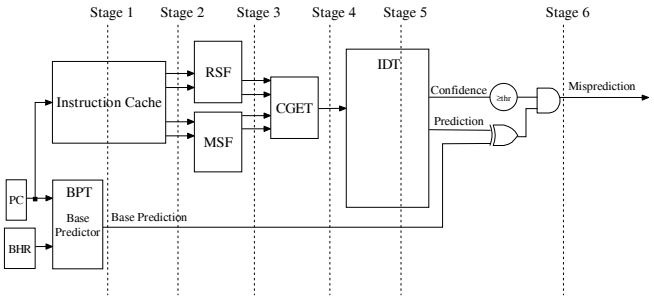

Figure 5. Pipeline with an Overriding Isomorphic-Based Branch Predictor

to decide whether to trust the prediction from the isomorphic-predictor. When the isomorphic-prediction is confident and the two predictions agree no action is taken, otherwise if they disagree the prediction is reversed. This approach is similar in spirit with other overriding predictors that have been proposed recently and shown to have promising performance [4, 17, 3].

The isomorphic-predictor is a variation of the method introduced in Section 3 and its logical organization is very similar to the one in Fig. 3 except it does not use an MSF. The RSF contains as many registers as the architectural registers (in this work 67).

The fields included in a signature are the OptypeSign, DataSign and SafeBit. The size for the OptypeSign and DataSign fields depends on the size of the IDT, and the number of bits in IDT tags. The CGET mechanism implements all transformations and features reported in Section 3. For unsafe-memory transformation the PCBHR is used.

The IDT is tagged with a hashed component-graph signature and is only read and updated by conditional branch instructions because we are interested only in detecting the isomorphism of those instructions. Each entry in the IDT also contains one bit that provides the branch direction prediction. The IDT index is computed by folding and hashing the OptypeSign and DataSign fields of a branch's signature. When the CBR transformation is applied during the encoding, indicated by a signal from the CGET unit, the prediction provided by the IDT is reversed.

A prediction by IDT is used only when there is a high confidence in the isomorphic-prediction. The confidence is based on a tag match and the value of a reseting counter maintained in each IDT entry. The counter is incremented when the isomorphic predictor gives correct predictions and reset otherwise. The counter is considered high-confident when its value is maximum.

The IDT is updated by a branch when is mispredicted by the base predictor or when its signature has a tag match in the IDT. When the IDT is associative and an entry needs to be evicted from a set, the entry with the lower confidence gets replaced.

To facilitate better confidence estimation, when a signature is inserted in the IDT its initial confidence is set to maximum when its safe-bit is set, otherwise it is set to minimum.

Fetch/Issue/Commit Width	4
Pipeline Stages	20
Instruction Window Size	128
LSQ Size	64
L1 I$	2cycle, 16KB, 64B blocks, 2-way
L1 D$	1cycle, 8KB, 64B blocks, 4-way
ALU/Memory Ports	4/2
L2$	7cycles, 512KB, 128B blocks, 8-way
Memory	200 cycles

Table 1. Out-of-Order Processor Parameters

5 Experimental Framework

To establish the usefulness of the IDM a simulation study was performed for the overriding branch predictor presented in Section 4 that includes an isomorphic-based predictor component.

The analysis was performed for an out-of-order superscalar processor with the attributes shown in Table 1. The simulator models accurately pipeline stages, speculative updates, execution from the wrong path, and misprediction recovery. The simulator was build on top of the simplescalar for the PISA architecture [2]. Experiments were performed with train or reference inputs for complete runs of SPEC95 integer benchmarks, and selected regions of a subset of the SPEC2000 benchmarks. We emphasize that the observations are indicative of the behavior for specific datasets and simulated regions.

The base predictor used is an idealized *gshare.fast* predictor [5] with no gap between the older and more recent branch history. This is a pipelined predictor that has an effective latency of one cycle.

The experimentation compares the performance of a 32KB *gshare.fast* predictor with the performance of a 16KB *gshare.fast* predictor combined with an IDM based predictor based on isomorphism.

The isomorphic-predictor is a two way-associative 16KB IDT. Each IDT entry includes a 4-bit tag, a 3-bit confidence and a 1-bit for prediction. This IDT configuration requires signatures that are at least 18 bits long. The 13 bits are needed to index the IDT, 4 bits are used for tag match and the remaining bit is the safe-bit. The RSF cost without any optimization must be at least 67x18 bits (151 bytes). The latency of the isomorphic-predictor is assumed to be 4 cycles and an isomorphic prediction is available 5 cycles after the base prediction. The number of branch history bits used for unsafe memory transformations are 6 for sp base register, 3 for gp, and 6 for $other$.

The performance of the proposed predictor was also compared against a configuration that employs only a fast 32KB hybrid base predictor proposed by McFarling [7] that consists of three tables: a 16KB gshare, an 8KB bimodal, and an 8KB selector. The latency for this predictor was assumed to be a single cycle. This latency is optimistic since it is not known how such a predictor can be pipelined and have a single cycle effective latency. This predictor is used to determine whether the isomorphic predictor captures branch behavior not captured by a bimodal predictor.

6 Results

Fig. 6 shows the speedup for two predictor configurations over a 32KB gshare predictor. The first column (GI) shows the performance of the proposed 16KB gshare predictor combined with a 16KB overriding IDM based predictor. For all benchmarks the proposed predictor achieves better performance as compared to a 32KB gshare predictor. In particular, for benchmarks *equake00*, *bzip00* and *twolf00* it achieves performance improvements of 7%, 6% and 5% respectively. This indicates that the IDM based predictor can be useful for improving performance.

To further establish the usefulness of the IDM based predictor the second column (H) shows the performance of a 32KB combining gshare/bimodal predictor with a single cycle delay. It can be observed that for several benchmarks - such as *comp95*, *twolf00* and *equake00* - the combining predictor's performance is significantly worse than the overriding. This demonstrates that performance can benefit more by adding a 16KB isomorphic overriding predictor to a 16KB gshare predictor, rather than adding a 16KB bimodal/selector. This also suggests that the isomorphic predictor learns branch behavior in a more effective manner than a bimodal predictor, and it gives correct predictions for branches that neither the gshare nor the bimodal predictors can capture.

Fig. 7 shows an analysis of branch prediction behavior for the overriding predictor. The branches are divided into four categories, according to the prediction made by the fast and the overriding predictors. BaseCor and BaseIncor indicate whether the branch was predicted correctly or incorrectly by the fast predictor. Similarly Over and NoOver indicate whether the branch was overridden or not overridden by the second level predictor. For most benchmarks, we notice that the percentage of branches that the overriding was incorrect (BaseCor-Over) is very small. This suggests that the confidence estimation employed in the isomorphic predictor is effective in preventing pseudo-isomorphism. From the data in Fig. 6 and Fig. 7 we can observe that benchmarks with 1% or more of their branches being overriden correctly (BaseIncor-Over), have at least a 2% performance improvement. However, for some benchmarks with large amount of correct-overrides, such as *gcc95* and *go95* the performance improvement is modest. This can be explained by considering that these benchmarks have small but still significant amount of incorrect overrides (BaseCor-Over). This indicates that to have significant performance improvements, it may be necessary to have a significant portion of branches correctly overridden (at least 1%) and a much smaller fraction of incorrectly overridden branches.

The performance potential of the isomorphic predictor is possibly limited by the confidence-selection mechanism used. Fig. 8 presents the percentage of branches for which the base predictor gives an incorrect prediction and the isomorphic overriding predictor could have overridden them correctly (Potential). The lighter portion of the column represents the fraction of these branches for which the isomorphic predictor actually overrides the base predictor (Captured). The difference between the potential and the captured represents the branches for which the overriding predictor disagreed with the base but did not override because of low confidence. From the data, we can notice that the current confidence-selection mechanism usually captures less than half of the potential. This may suggest the need for a more accurate confidence-selection mechanism.

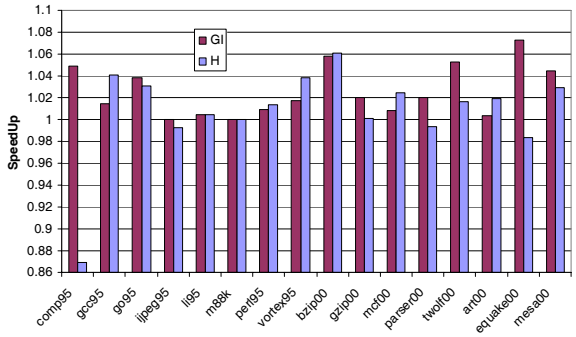

Figure 6. SpeedUp over a 32KB *gshare.fast* Predictor

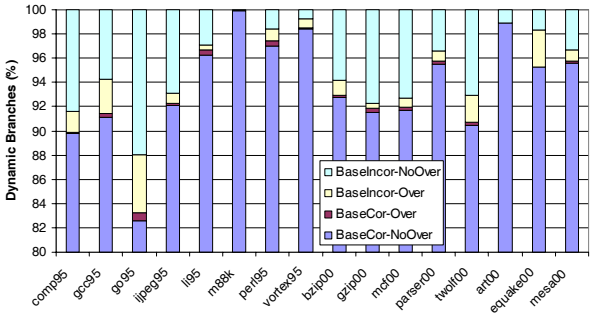

Figure 7. Breakdown of Prediction Behavior with an Overriding Predictor

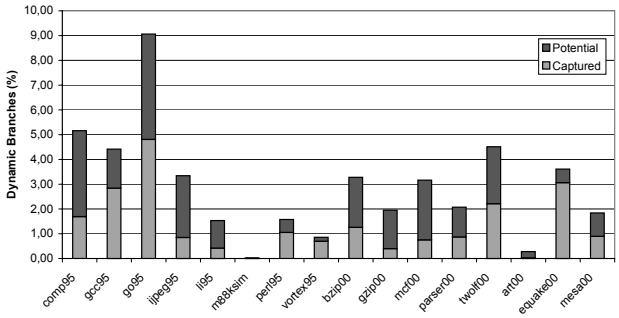

Figure 8. Mispredictions not Overridden due to Confidence Estimation

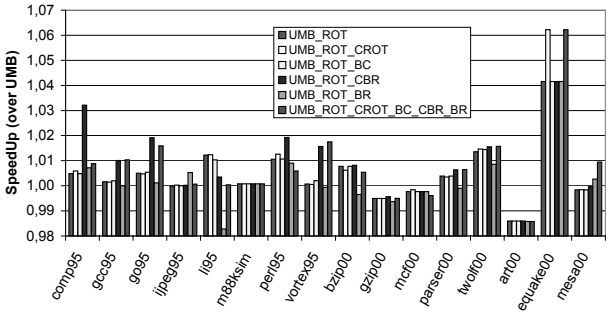

Figure 9. SpeedUp with various combinations of transformations

Fig. 9 highlights the importance of the various transformations employed and of the rotation used during encoding. The data show the speedup obtained with a configuration using a particular mix of transformations over a configuration that employs only the UMB transformation. The results show that using transformations, in addition to unsafe memory bypassing, is beneficial to performance for almost all benchmarks. The data also show that the amount of improvement and the configuration with the best performance varies across benchmarks. In most cases the configuration that combines all the transformations together offers also the best or close to the best performance.

The CBR transformation appears to be the most useful transformation in many benchmarks. This indicates that often the base predictor mispredicts on branches that are isomorphic to other branches that simply test complementary conditions. For one benchmark, *equake00*, the CROT transformation contributes significantly. This implies that the base predictor mispredicts branches that are isomorphic to other branches that have the same inputs but in different order.

The data also show that for some benchmarks using a transformation can be detrimental to performance. This mainly occurs because the combination of unsafe memory bypassing with other types of transformations can result in pseudo-isomorphism. This is illustrated with the aid of an example in Fig. 10. The two original component-graphs are non-isomorphic but with the application of the BR transformation they become pseudo-isomorphic (the value loaded from memory is different in the two components).

Overall, although the combination of the various transformations appears beneficial, the amounts of improvement are modest. We have evidence that suggest that this can be improved by applying safe memory bypassing. The components with unsafe memory bypassing are typically small - rarely more than 10 instructions - with little opportunity for isomorphic transformations. By applying safe memory bypassing components can go past loads and therefore both the size of the components and the opportunity for isomorphism will grow. We have developed two accurate mechanisms for memory bypassing from the stack and the memory segment and we are currently exploring a method for memory bypassing in the heap. Our future work will report on these mechanisms.

7 Related Work

Previous work [16, 10] proposed mechanisms for encoding data-dependence graphs on the fly and predicting values based on encoded graph information. The mechanism in [16] was used in a subsequent work to predict [15] conditional branches. These earlier mechanisms resemble the one introduced in Section 3, however, they were not geared toward facilitating or detecting isomorphism. Specifically, the dependence-graphs considered in the above work were limited to include only dynamic instructions that are currently in the instruction window of a processor. The dependence-graph for instructions that already committed is represented with the values they produced and stored in registers. One other difference is that these previous works did not employ transformations to remove superfluous instructions. Memory bypassing is mentioned and its ideal potential is evaluated in [10],

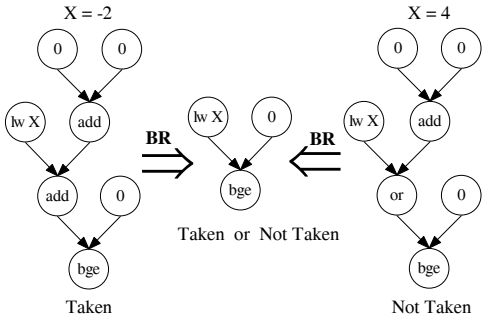

Figure 10. Example where transformations convert Non-Isomorphism to Pseudo-Isomorphism

however, no concrete scheme is proposed for implementing it. Nevertheless, three practical options for encoding *unsafely* load instructions were proposed: (a) predicting their value [16], (b) using the encoding of their address graph [16, 10], and (c) using the PC of the encoded load [16].

One other related work is a mechanism proposed by [3] for tracking data dependences. This mechanism can be used to track the registers that affect an instruction, but not past load instructions. Data dependence tracking was applied to branch prediction. The basic idea is to use the tracking mechanism to determine the registers that affect a branch instruction and use known register values and the mask of registers that affect a branch, to index and train a branch predictor. This mechanism may be limited: (a) by not considering dependences past loads, and (b) by ignoring the information about the instructions and their dependences that use the values selected by this mechanism to compute the branch direction.

Recently, a method was proposed that uses the dynamic data dependences to determine the branch history that a branch instruction should be correlating on [17]. In our view, this method attempts indirectly to encode the data dependence graph that affects a branch. Such an indirect encoding may be limiting the potential of this approach. One other possible limitation of this scheme is that memory dependences are always unsafely transformed since for loads instruction the address dependence is considered.

8 Conclusions and Future Work

This paper proposes IDM: a hardware-based method for detecting instruction-isomorphism dynamically. The proposed method implements several dynamic data dependence graph transformations.

A method that can detect instruction-isomorphism can have several uses and this paper consider its application to branch prediction. An overriding branch predictor is proposed that includes an isomorphic-based branch predictor component that is a variation of the IDM. Experimental analysis using SPEC benchmarks shows that the proposed prediction approach can be beneficial to performance.

This paper points to several directions of future work: develop other component-graph transformations, improve the memory-bypassing transformation, design a more accurate confidence estimator, and perform a more comprehensive comparison of the proposed predictor with other known branch predictors. Another direction of research is to consider other microarchitectural applications of the instruction-isomorphism detection method.

9 Acknowledgments

The authors would like to thank the reviewers for their constructive critique and acknowledge Intel for supporting our research activity.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2003.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[3] L. Chen, S. Drposho, and D. H. Albonesi. Dynamic Data Dependence Tracking and its Application to Branch Prediction. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 65–76, February 2003.

[4] D. A. Jimenez. Fast Path-Based Neural Branch Prediction. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 2003.

[5] D. A. Jimenez and C. Lin. Reconsidering Complex Branch Predictors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 43–52, February 2003.

[6] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Data Speculation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[7] S. McFarling. Combining Branch Predictors. Technical Report DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[8] A. Moshovos. Memory Dependence Prediction. *PhD Thesis, University of Wisconsin-Madison*, Dec. 1998.

[9] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[10] Y. Sazeides. Dependence Based Value Prediction. Technical Report CS-TR-02-00, University of Cyprus, February 2002.

[11] Y. Sazeides. Instruction-isomorphism in program execution. *The Journal of Instruction Level Parallelism*, 5, November 2003.

[12] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[13] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 194–205, June 1997.

[14] A. Sodani and G. S. Sohi. An Empirical Analysis of Instruction Repetition. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, October 1998.

[15] R. Thomas. Computation History Based Prediction. *PhD Thesis, University of Maryland - College Park*, 2003.

[16] R. Thomas and M. Franklin. Using Dataflow Based Context for Accurate Value Prediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 107–117, September 2001.

[17] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 314–323, June 2003.

[18] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.