Journal of Instruction-Level Parallelism 5 (2003) 1-22 Submitted 9/03; published 11/03

Instruction–Isomorphism in Program Execution

Yiannakis Sazeides YANOS@CS.UCY.AC.CY
Department of Computer Science
University of Cyprus
Nicosia, Cyprus

Abstract

This paper identifies a fundamental runtime program property: *Instruction–Isomorphism*. An instruction instance is said to be isomorphic if its component - information derived from the instruction and its backward dynamic data dependence graph - is identical to the component of an instruction executed earlier. By definition an isomorphic instruction will produce exactly the same output with the earlier instruction.

This work introduces a taxonomy of isomorphic behavior, describes transformations that can change the isomorphic behavior of an instruction, characterizes empirically various aspects of instruction–isomorphism and suggests directions for improving predictors and performance.

The empirical analysis shows that there is very little instruction–isomorphism in the dynamic dependence graph of a program. This is due to programming conventions and architectural semantics that introduce a lot of "overhead" nodes and dependences. However, by transforming the dynamic dependence graph closer to its dataflow form, instruction–isomorphism becomes prominent. The data show that for SPEC benchmarks, depending on the benchmark and dataset, 65 to 99.9% of the dynamic instructions are isomorphic.

1. Introduction

There is a plethora of work showing that dynamic program information exhibits predictable or repetitive behavior. Specifically, (a) several basic types of information have been shown to be predictable: branch directions [1], branch targets [2], memory data addresses [3], values [4, 5] and dependences [6], (b) values produced by instructions were shown to repeat from a small set of values [4], and (c) instructions were demonstrated to often repeat with the same input and output value [7].

These phenomena may be caused by "inefficiencies" at various computational layers: algorithmic, programming, compiler and architectural. The philosophy that prevails in the research community is to exploit these phenomena with optimizations at computational levels below programming.

The predictability or repetition of a particular information type is measured in terms of a specific mechanism. The organization and policies of such a mechanism are aimed to detect program behavior that results in predictable or repetitive computation. Therefore, for a mechanism to be successful it is important to capture accurately typical program behavior. However, proposed mechanisms are usually based on adhoc models of program behavior or are adaptations of existing mechanisms from other areas.

Although successful mechanisms for identifying predictable [1, 3, 4, 6] or repetitive [7, 8] computation are available, the lack of a theoretical basis for modeling program behavior may prevent the development of better mechanisms. Consequently, a more basic research of program runtime behavior may be needed for an understanding of how execution manifests itself into predictable and repetitive behavior.

This paper provides a step in this direction: it identifies *instruction–isomorphism* as an esoteric runtime program property. An instruction instance is said to be isomorphic if its component - information derived from the instruction and its backward dynamic data dependence graph - is identical to the component of an instruction executed earlier. Therefore instruction–isomorphism occurs when there is repetition in the dynamic structure of the program. By definition, an isomorphic instruction will produce exactly the same output as the earlier instruction. Previous work [9, 7] established that predictability and repetition is influenced more by the structure of the program and immediate values and less by the input data. This work builds on this observation, because the component of an instruction represents program structure that can influence the outcome of the instruction. Provided that instruction–isomorphism is a dominant program execution phenomenon, information derived from an instruction component may represent a useful source of information to detect predictable or repetitive computation.

Contributions

What differentiates this paper from earlier work is the scope and methods used to investigate how program structure influences instruction–isomorphism: the scope is more holistic because it can consider the entire dynamic dependence graph leading to an instruction, and the methods are more comprehensive since they include various transformations on the dependence graph to facilitate isomorphism. The paper also includes an empirical characterization and analysis that illuminates various aspects of instruction–isomorphism for SPEC benchmarks. We believe that these differences are sufficient to provide new insight and direction on how to deal with inefficiencies observed during program execution.

Outline

Section 2 discusses related work. Section 3 introduces formal definitions for instruction–isomorphism. Section 4 describes the simulation framework used in this work and results are presented in Section 5. We provide conclusions and direction for future research in Section 6.

2. Related Work

Instruction–Repetition, Reuse and Isomorphism: The most related concepts to instruction–isomorphism are instruction–repetition and instruction–reuse [10, 7, 11]. Instruction–repetition [7] considers repetition of input–output values at instruction granularity. Instruction–reuse is a microarchitectural method motivated by repetition. The most general reuse scheme [10] learns for each dynamic instruction (a) its backward dynamic dependence graph with respect to other instructions in its fetch group, (b) livein values to this dependence graph, and (c) an output value. If the dependence graph and livein values for an instruction repeat, the learned outcome can be reused without having to execute the instructions in its dependence graph. It is expected that reuse can exploit only a subset of instruction–repetition due to the stricter criteria for detecting reuse.

The backward dynamic dependence graph and livein values, as used in instruction–reuse, define a component for an instruction instance. In this respect, when instruction–reuse occurs it corresponds to a case of isomorphic behavior. In general, however, instruction-isomorphism and instruction-reuse are not the same.

If we view the various proposed reuse techniques as transformations of the dependence graph that expose isomorphism, then the following distinguish this paper. As far as we know this is the first study to consider entire backward dynamic dependence graphs for detecting repetition. In contrast, all previous reuse work have considered only part of the dependence graph, constrained either by microarchitectural parameters [10, 8] or compiler analysis [12, 13], and thus may had limited scope. Furthermore, virtually no previous study consider restructuring and/or relabeling the dependence graph to facilitate more isomorphism without introducing new values nodes (see Section 3.3.2). Transformations that introduce arbitrary livein value nodes, such as in previous reuse work, may not be desirable because they preserve less program structure and are more likely to depend on microarchitectural parameters and events. For example, instruction–reuse may learn multiple reuse scenarios for an instruction depending on its position in fetch groups, and may be hindered by the unavailability of livein values. In contrast, instruction–isomorphism relies mainly on architectural information.

Some previous work considered transforming the dependence graph [14, 15, 16] to enable reuse of computation without a PC (program counter) match. In [14] two cases of reuse behavior are distinguished *quasi-invariant* and *quasi-common-subexpression*. These are roughly analogous to *name* and *type* labeling defined later in this work.

We note that the work in this paper is an extension of an earlier report [17].

Isomorphism: Graph isomorphism was used to detect redundancy in a program. Komondoor and Horwitz [18] identify source code duplication by checking for isomorphism in program slices extracted from the static program dependence graph. The motivation is to enable better software development by replacing duplicated source code to a single macro or a call to a separate procedure.

Larus and Chandra [19] used isomorphism to identify redundant common subexpressions to tune compilers. Their method maintains for each machine register, during the execution of a program, a *portion* of the dynamic data dependence graph that lead to its most recent definition. Redundancies are determined by checking if a graph for a new definition is isomorphic with any of the current graphs in the registers. If isomorphism is detected, then computation that lead to the new definition is redundant and available from another register.

The type and amounts of isomorphism reported in [19, 18] cannot be used in a systematic way to assess instruction–isomorphism. The objective of our work is not to detect and/or eliminate redundancy but to characterize repeating structural patterns during execution - which may not be redundant. Nevertheless, the hypothesis and observations in these two studies provide a basis for explaining possible causes of instruction–isomorphism.

Slicing: Slicing [20] is a method useful for determining the statements in a program that may have an effect on another statement. Slicing can be applied backward or forward and to the static or dynamic dependence graph of a program [20, 21, 22, 23]. Typically a slice, irrespective of its direction and graph type, corresponds to a subgraph of the program dependence graph. A program dependence graph [24] is a static program graph with control and data dependences. This means that a slice, static or dynamic, will contain at most a single instance of an instruction. Few papers [19, 7] departed from this canonical notion of a slice and refer to a subgraph of the dynamic dependence graph [22] that leads to an instruction instance also as “dynamic slice”. Such a “dynamic slice” is acyclic and may include multiple instances of an instruction. In this paper, we investigate backward “dynamic slices”. To avoid any possible confusion, we refer to these slices as **components**.

3. Instruction–Isomorphism

This section provides a definition for instruction–isomorphism and discusses isomorphism relevant issues.

3.1 Definitions

A dynamic instance, *i*, of an instruction defines a component, C_i. The component of an instruction instance includes the instruction and its dynamic backward dependence graph. In particular, the component of an instruction instance is a directed acyclic graph that contains: (a) the instruction itself as its root node, and (b) edges to all its direct predecessor components. The direction of edges is from consumer to producer. Therefore the component of an instruction includes all instruction instances (nodes) that the instruction has direct or indirect dependence on. Because a component is derived from the dynamic dependence graph it may include multiple instances of an instruction.

A component, C_i, has a value, $V(C_i)$, which is the value produced by the instruction instance *i*.

The set of nodes that are direct predecessors to another node, *i*, define an ordered set called the *predecessor set* of *i*. This is the set of instruction instances an instruction directly depends on. The ordering of predecessor sets is identical for nodes with same labels. This is required to avoid detecting isomorphism when operators are not commutative and/or associative[1].

Component edges are not labeled but nodes are. The labeling information is such so that it uniquely identifies the computation to be performed by a node. The PC and opcode of an instruction is sufficient.

Values read by instructions but not produced by program instructions are represented as nodes with the PC set to the value and the opcode to a special opcode indicating a value node. These nodes are needed to distinguish components that perform the same computation with different input values.

The *depth* of a component is the longest path from its root. Depth does not include value nodes.

Although the above definition is applicable to components that include both control and data dependences, the remaining paper considers components with only data dependences. The rationale for ignoring control dependences is discussed in Section 3.3.1.

Structure and Relation of Components

An instruction can be classified into one of the following four sets depending on the structure of a component and its relation with components of other instructions:

maximal, if its component is not included in any other component,

source, if it does not depend on any other instruction and consequently its component depth is one,

source–maximal, if its both *source* and *maximal*. The previous two categories exclude the instructions from this set. The sum of *source* and *source–maximal* is the set of all instructions with depth one, and

middle, if its none of the above, i.e. lies in the middle of at least one component.

3.2 Isomorphism Basics

Lets consider two dynamic instruction instances, *i* and *j*, and their corresponding components, C_i and C_j. Two components are isomorphic if the labeling of the two root nodes, *i* and *j*, is the same, and each ordered pair of nodes in their predecessor sets is isomorphic. Effectively, two components

1. this may be pessimistic for some instruction types

are isomorphic if they have exactly the same shape and labeling. Therefore when comparing the two components one of the following behaviors will be observed:

isomorphic equality: if the two components are isomorphic then the two components should produce the same value, V(C_i)=V(C_j),
otherwise they are non-isomorphic and either of the following will occur
non-isomorphic inequality: the values are unequal, V(C_i)$\neq$V(C_j), or
non-isomorphic equality: the values are equal, V(C_i)=V(C_j).

Isomorphic Equality: occurs when two instructions are isomorphic. Therefore their component sizes are equal and each instruction in the one component is isomorphic to an instruction in the other component.
Non-isomorphic Inequality: is caused by instructions with dependence structure that has not been observed before. Although, it is unclear how to convert components with this behavior to isomorphic, it may be possible to represent them "compactly". This may be achieved by using functions, that depending on the structural difference between two components can compute the difference in their output values.
Non-isomorphic equality: represents an important case of isomorphic behavior because it may be possible with transformations to convert components from non-isomorphic to isomorphic. There are at least three classes of transformations to consider (a) *restructuring* components without introducing value nodes, (b) *relabeling* nodes, and (c) *replacing* portion of a component with a value node. Transformations can be further classified into *safe* and *unsafe* depending on how they affect components outputs. A transformation is safe if the execution of the original and transformed component produce the same output value, otherwise it is unsafe. Unsafe transformations may lead to components with incomplete information where the following isomorphic behavior can happen:

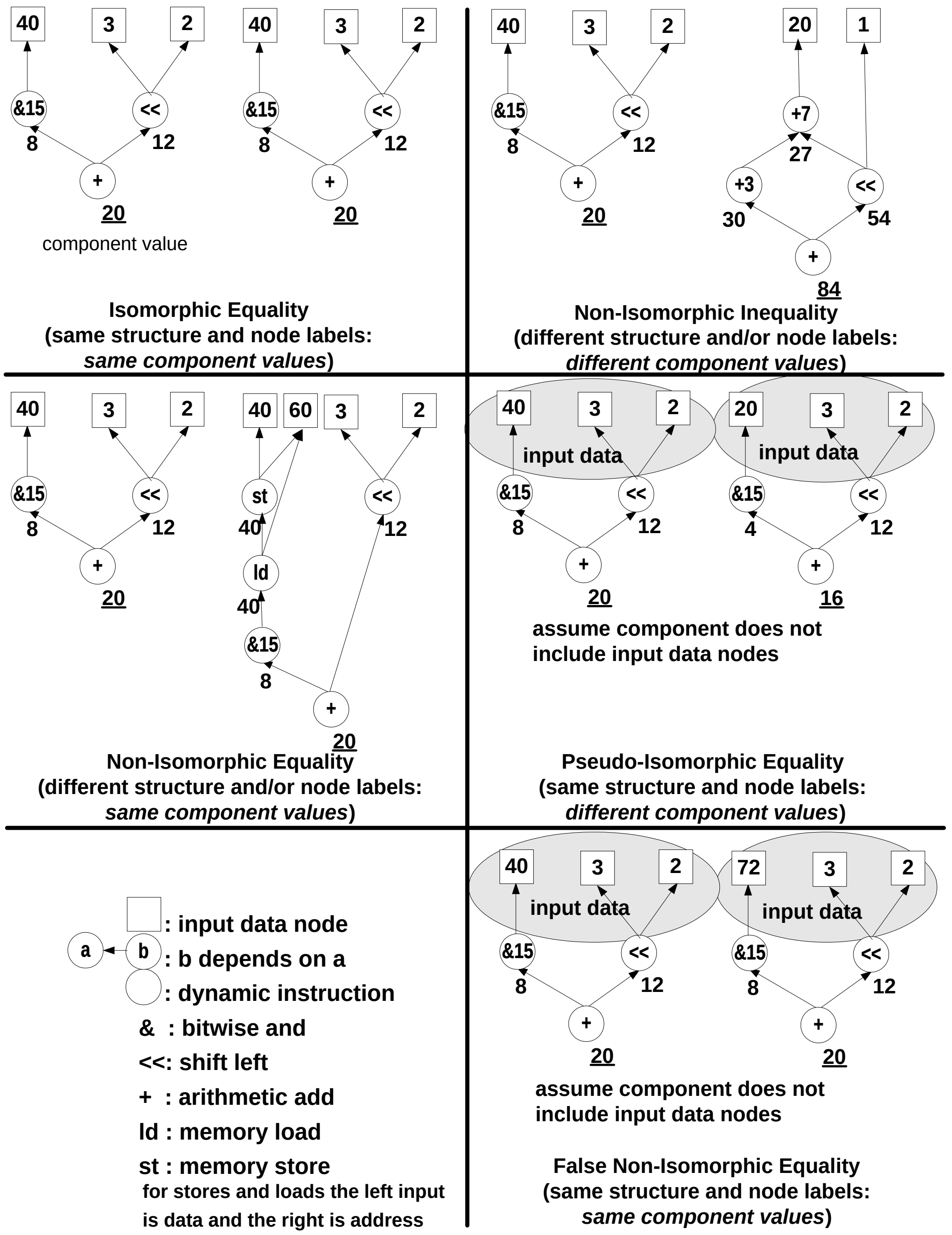
pseudo-isomorphic equality: occurs when two components are isomorphic but their output values are not equal. Analysis of the sensitivity of isomorphism to unsafe transformations can reveal whether exact component information is necessary to detect isomorphism,
false non-isomorphic equality: occurs when prior to the unsafe transformation the two components exhibited non-isomorphic equality but with incomplete information become isomorphic equal. This is analogous to constructive aliasing in predictors [25] and should be useful to characterize its frequency.

The various cases of instruction–isomorphic behavior are presented pictorially in Fig. 1. For each case two components are compared.

Non-isomorphic equality and transformations are discussed in more detail in the next subsection. Safe transformations is the focus but unsafe transformations are also discussed. Non-isomorphic inequality, is not examined and should be the subject of future work.

3.3 Non-Isomorphic Equality implies Isomorphic Computation is Suppressed?

If all program dependences - both control and data - were included in instruction components, very likely each component would have had either unique pattern or size, and isomorphic–equality would have been a rare phenomenon. Combining this assumption with the observation that instructions produce very often the same values [4, 26, 7], suggests that the dominant program isomorphic be-



40
3
2
&15
<<
8
12
+
20
component value
Isomorphic Equality
(same structure and node labels:
same component values)
Non-Isomorphic Inequality
(different structure and/or node labels:
different component values)
20
1
+7
27
+3
<<
30
54
84
60
st
ld
40
input data
4
16
assume component does not
include input data nodes
Non-Isomorphic Equality
(different structure and/or node labels:
same component values)
Pseudo-Isomorphic Equality
(same structure and node labels:
different component values)
72
: input data node
a
b
: b depends on a
: dynamic instruction
& : bitwise and
<<: shift left
+ : arithmetic add
ld : memory load
st : memory store
for stores and loads the left input
is data and the right is address
False Non-Isomorphic Equality
(same structure and node labels:
same component values)

Figure 1: Instruction–Isomorphic behavior that can occur when comparing two components

havior would be non-isomorphic equality. However, some program dependences are manifestations of architectural semantics that would not exist in a dataflow form of computation. For example, data movement nodes and memory address calculations are unessential since they are used to move data from one location to another. Therefore, isomorphic-equality may exist in program execution but is suppressed, in the form of non-isomorphic equality, and to expose it requires transformations on the dependence graph.
Objectives of Transformations: we believe that the main objectives of *most* transformations should be to increase isomorphism by converting components into a more concise representation without introducing value nodes. Conciseness is useful in making isomorphic-equality more likely to occur - smaller components are more likely to match. By not introducing value nodes we rely more on the structure of the program and less on microarchitectural parameters (see Section 2). Although this study is not evaluating uses of isomorphism, we believe the former to be more robust basis to “build on”. Changes in the isomorphic behavior of an instruction are feasible through transformation of its component structure. Thus transformations may change a component’s size, depth, and its relation with other components.

Restructuring, *relabeling* and *replacing* transformations, introduced in the previous section, are discussed next. Note that this set of transformations may not be complete in that other transformations, to be determined, may uncover additional instruction–isomorphism.

3.3.1 RESTRUCTURING WITHOUT INTRODUCING VALUE NODES

This class of transformations rearranges edges without introducing new value nodes, to eliminate edges and nodes that are nonessential. The following are examples of safe-restructuring-transformations:
NoAddr: eliminate the edges for address operands of load and store instructions. Effectively store and load nodes have one edge leaving them that corresponds to the value that will be written/read from memory. This is *safe* because address operands are only useful to specify where data are stored/read from, and do not affect the value that is transferred.
BypLdSt: bypass nodes that move data from/to memory (instances of loads and stores) so that consumer nodes are linked directly to their producers. Without this transformation two components can be classified as non-isomorphic-equal because of one more move of a value. Loads and stores are not eliminated from the graph just the edges that emanate from them. Loads and stores are preserved so that we can investigate their isomorphic behavior regarding the computation patterns that lead to them.
BypMov: bypass a node that moves data between two registers. This transformation mainly can eliminate dependences to overhead computation due to call/return convention.
BypComp: bypass two dependent nodes when the one adds(subtracts) an immediate value whereas the other subtracts(adds) the same immediate value. Effectively, such a sequence corresponds to a move since the input value of the first instruction is the same as the output of the second. This can be determined by only checking the opcodes of dependent instructions. When using this transformation, an instruction that originally had a dependence on the second instruction will be linked to the node producing the input of the first instruction.

Although this is a general transformation, it is mainly aimed to reduce the number of unique components defining the stack pointer (SP). Each SP definition typically depends on a previous SP definition. Thus SP updating instructions can form a very long chain of dependent instructions each defining a unique component. What is more, the instructions that depend on SP computation will also define unique components leading to more unique behavior and less isomorphism. However, SP defining instructions typically appear in pairs at the prologue and epilogue of functions decrementing and incrementing, respectively, the SP by the same amount. And therefore are amenable to this transformation.

NoControl: control dependences are critical part of the program structure and highly influential for the shaping of component graphs. They can provide precise information about how the component of an instruction was formed and the ordering of component execution. However, once a component is formed, is safe to remove control dependences because they do not influence the dataflow in the component. *Therefore, this work does not consider isomorphic behavior in the presence of control dependences in components*. A consequence of the elimination of control dependences is that all components corresponding to instances of conditional transfer instructions have no data dependent successors. The value for a conditional branch component is defined to be the branch direction.

Virtually each time any of the above transformations is applied it converts an instruction to maximal or source-maximal. This is the case because the dependence to the output of an instruction is eliminated which means the component of the instruction is not included in any other component. Another ramification of the above transformations is a reduction in the size and depth of a component.

Fig. 2 illustrates how the different transformations change the structure of an example dependence graph. After a number of transformations have been applied on the *original dependence graph*, that eliminate or rearrange edges, the *transformed graph* is produced. The unique computational patterns (unique components) that remain after the transformations are shown in the *unique components* graph. Note that all computations in the original graph can be mapped to a component in the unique components graph.

All of the above safe transformations aimed to convert non-isomorphic equality to isomorphic equality. *Unsafe transformations*, arbitrary or systematic removal of nodes and edges from components, can lead to pseudo-isomorphic and false-isomorphic equality. When applying unsafe transformations, the amount of pseudo-isomorphism and false-isomorphism produced can be illuminating as to whether exact component information is necessary to detect isomorphism. Also it may be useful for understanding the trade-off between safe and unsafe transformations.

A special case of unsafe transformation is to remove all value nodes and edges leading to them. Analysis of isomorphism in this situation can provide insight as to how sensitive is computation to input data. In Fig. 1, pseudo-isomorphism and false non-isomorphism occur as a result of the transformation that removed the dependences to input data.

This work does not consider mechanisms for accomplishing the various transformations. It is mainly concerned with their potential to increase instruction–isomorphism. Future work needs to consider practical schemes for facilitating and detecting isomorphism. Several other restructuring transformations, safe and unsafe, exist but are beyond the scope of this paper.

In Section 5, is established empirically how significant the various safe transformations are on isomorphism and component structure.

3.3.2 RELABELING NODES: NAME AND TYPE ISOMORPHISM

It is apparent from the discussion so far that the amount of isomorphism that can be observed is influenced by the edges and nodes that are included in components. However, one other parameter influences the amount of observed isomorphism: node labeling.

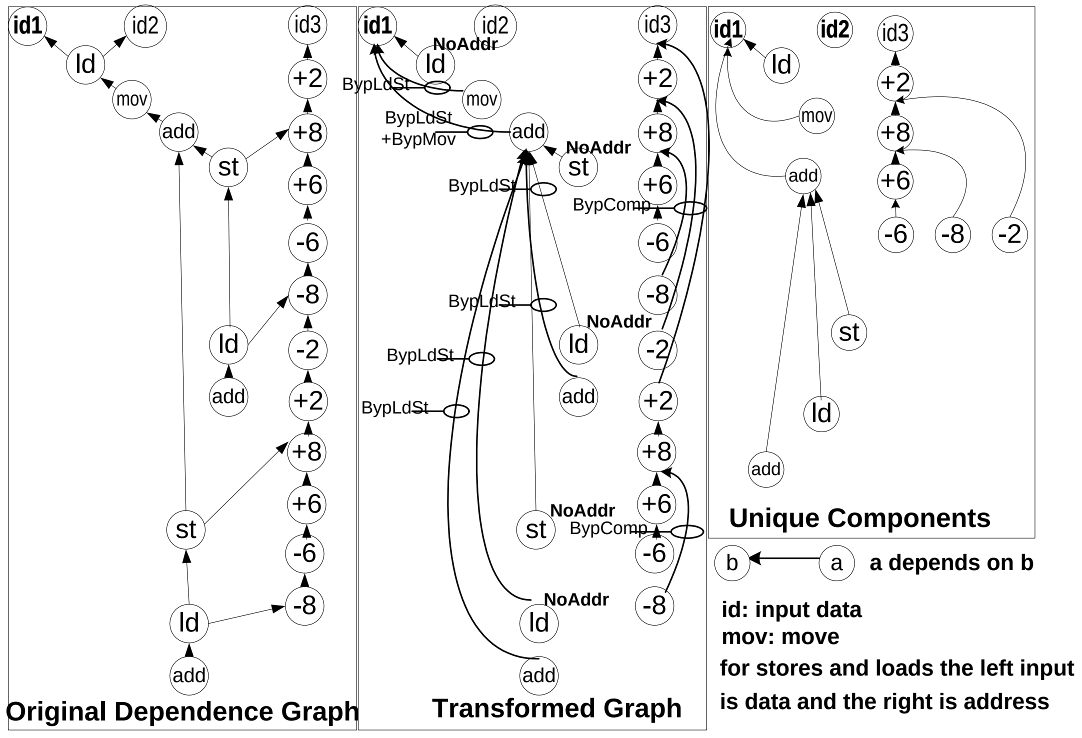


Figure 2: Effects of Transformations

Two cases of safe labeling are presented: *name* and *type*. With name–labeling the node label includes the PC and the opcode of an instruction. This information is sufficient for safe labeling but may be conservative. An alternative labeling is one that includes the optype and immediate value of an instruction. This is referred to as type–labeling. Value nodes with type–labeling are indicated with a special optype and their value is assigned to the immediate field of the label. With type-labeling one node may represent different PCs with the same instruction type and immediate value, and therefore isomorphism detected with type–labeling will be a superset of the isomorphism detected with name-labeling. Type-labeling can be viewed as another transformation of a component graph towards a dataflow graph since the PC is an architectural side-effect not needed in dataflow computation. Type–labeling can convert non-isomorphic equality to isomorphic equality when the same computational pattern is produced in different paths of a program. Note that type-labeling was used in all figures in the paper so far.

The potential benefit of type over name labeling is demonstrated with the aid of Fig. 3. This figure shows two components for instructions $eval + 4268\ subu\ \$21, \$2, \$3$, and $eval + 46d0\ subu\ \$21, \$2, \$3$, from an execution of benchmark *256.perl*. The two components have the same structure but three nodes have different PC labels. Therefore with name-labeling the two components would have been classified as non-isomorphic. However, the three nodes with different PC, perform the same computation. Therefore with type-labeling the components would have been isomorphic.

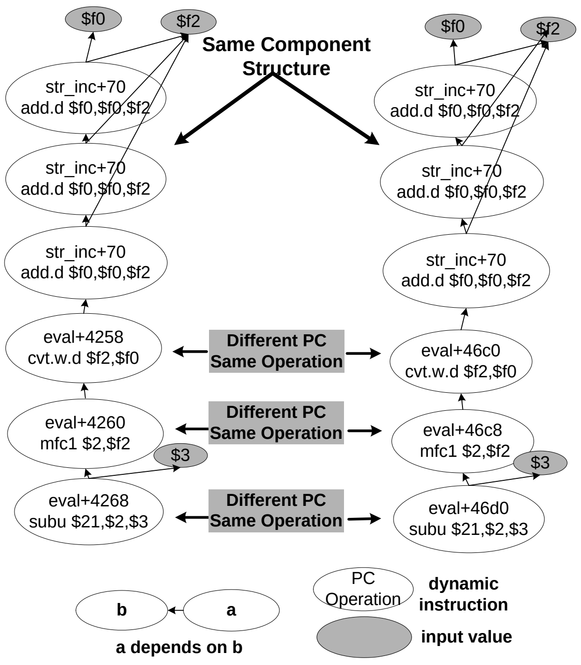

$f0
$f2
Same Component Structure
str_inc+70 add.d $f0,$f0,$f2
str_inc+70 add.d $f0,$f0,$f2
str_inc+70 add.d $f0,$f0,$f2
eval+4258 cvt.w.d $f2,$f0
eval+4260 mfc1 $2,$f2
$3
eval+4268 subu $21,$2,$3
Different PC Same Operation
Different PC Same Operation
Different PC Same Operation
$f0
$f2
str_inc+70 add.d $f0,$f0,$f2
str_inc+70 add.d $f0,$f0,$f2
str_inc+70 add.d $f0,$f0,$f2
eval+46c0 cvt.w.d $f2,$f0
eval+46c8 mfc1 $2,$f2
$3
eval+46d0 subu $21,$2,$3
b
a
a depends on b
PC Operation
dynamic instruction
input value

Figure 3: Name vs Type Labeling

Relabeling can also be *unsafe*. An example of unsafe labeling is one where nodes do not contain any label. In such a case is possible for nodes corresponding to different "computations" to be indistinguishable and thus lead to pseudo-isomorphic or false non-isomorphic equality.

This work only examines safe relabeling and in Section 5 results for *name* and *type* labeling are reported. Other types of safe and unsafe relabeling may exist and future work should investigate them.

3.3.3 REPLACEMENT: CONVERTING PORTION OF A COMPONENT TO A VALUE NODE

Replacing part of a component with a value node can enable more isomorphism. This may be desirable in cases where the number of unique components for the various instances of an instruction are much greater than the number of unique values the instruction produces. For example, non-linear instructions (*and*, *or*, *xor*) tend to have a smaller target than domain.

For a replacement to be safe, the definitions that emanate from nodes in a "replaced" subcomponent should be replaced with value nodes. Taking component replacement to the extreme, each component can be reduced to an instruction node with value nodes for each input. This represents the case with the highest amount of isomorphism (this is what instruction repetition measured in [7]), however it completely removes program dependence structure and this may be undesirable (Section 3.3). Therefore, *replacement* methods are faced with the following trade-off: amount of isomorphism they can achieve, and program structure they preserve. So far, no work has considered applying replacement "selectively" (where is likely to have higher payoff) or unsafe replacement (e.g. replace with a value node that may be unsafe times). Fig. 4 shows how replacement can be used to safely transform one component with a value node.

Although an important subject, replacement is not examined in this paper.

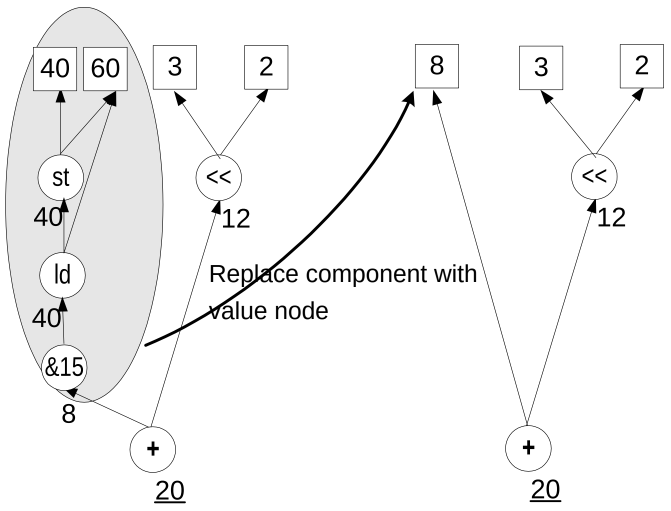


Figure 4: Safe Replacement of a Component with a Value Node

4. Simulation Framework

To characterize the isomorphism in program execution, a simulation study was performed for the simplescalar PISA architecture [27]. Measurements were performed with train or reference inputs for complete runs of SPEC95 benchmarks, and selected regions of SPEC2000 benchmarks. Table 1 shows the benchmarks, the number of instructions skipped before starting detailed simulations, the number of dynamic instructions simulated and the number of static instructions executed. The benchmarks were compiled using the simplescalar *gcc* compiler with *-O3* optimization. We emphasize that the observations are indicative of the behavior for specific datasets and simulated regions.

Arithmetic mean was used for reporting averages. Results are reported in terms of dynamic instructions or in terms of instructions that define unique components. The latter correspond to the dynamic instructions that their components have not been observed before and are also referred to as non–isomorphic instructions.

5. Results

5.1 Overall Isomorphism in Program Execution

Fig. 5 shows the degree of isomorphism for six cases for each benchmark. Each of the six bars corresponds to a specific combination of transformations and labeling. The first five bars were obtained using name–labeling and the sixth with type–labeling. The *All–Name* bar represents the case where all data dependences are included and name–labeling is used. The degree of isomorphism corresponds to the average number of times dynamic instructions occur with the same dependence graph leading to them (degree of one represents the case where there is no isomorphism). The inverse of the degree of isomorphism provides the fraction of dynamic instructions that define unique components. For example, when the degree of isomorphism is 50 it means that 2% of the dynamic instructions define unique components and that 98% of the instructions are isomorphic to instructions in the other 2%.

Benchmark	Skip(mil)	Dynamic Instr (mil)	Static Instr
compress95 INT	-	37	3446
gcc95 INT	-	178	112301
go95 INT	-	132	52347
ijpeg95 INT	-	129	15051
li95 INT	-	202	6441
perl95 INT	-	40	15385
vortex95 INT	-	101	59370
bzip00 INT	315	100	1734
gcc00 INT	700	100	1684
gzip00 INT	300	54	1529
mcf00 INT	2000	100	496
parser00 INT	400	100	1226
twolf00 INT	100	100	10686
vortex00 INT	100	100	13987
ammp00 FP	2000	100	1427
art00 FP	50	100	441
equake00FP	1300	100	838
mesa00 FP	350	40	3432

Table 1: Benchmark Characteristics

The results show that when all data dependences are considered, bar *All-Name*, very little isomorphism is found across all benchmarks. However, the data show that isomorphism can become prominent by employing various transformations. Specifically, when address dependences are removed, bar *NoAddr-Name*, there is a dramatic increase in isomorphism for most benchmarks. This indicates that although the dependence graph leading to address computation can be different in two components, frequently the remaining component structure is isomorphic. When address dependences are ignored and load-stores are bypassed, bar *NoAddr+BypLdSt-Name*, isomorphism is increased further. This indicates that components, without address dependences, may appear different due to operand movement through memory. The bypassing of register move operations, in addition to the *NoAddr and BypLdSt* transformations (bar *NoAddr+BypLdSt+BypMove-Name*), causes an additional increase in isomorphism. This suggests that components without address dependences and memory bypassing, can appear different due to data movement through registers. When the above transformations are coupled with bypassing of computation that corresponds to a move, there is even more increase in isomorphism, bar *NoAddr+BypLdSt+BypMove+BypComp-Name*. *BypComp* is important for benchmarks with a lot of call-return activity due to SP computations. Finally, when all the previous transformations are combined with type–labeling, bar *NoAddr+BypLdSt+BypMove+BypComp-Type*, there is yet more increase in isomorphism. This indicates that programs can perform computation with the same structure on different paths.

With respect to individual benchmark behavior, it can be observed that isomorphic behavior varies across benchmarks and that not all benchmarks exhibit similar sensitivity to the different transformations. The two compression benchmarks, *gzip00* and *compress95*, and *mesa00* exhibit low isomorphism and small sensitivity to the different transformations. The highest isomorphism is achieved with *li95*, *gcc00* and *ammp00*. These benchmarks are very sensitive to the transformations

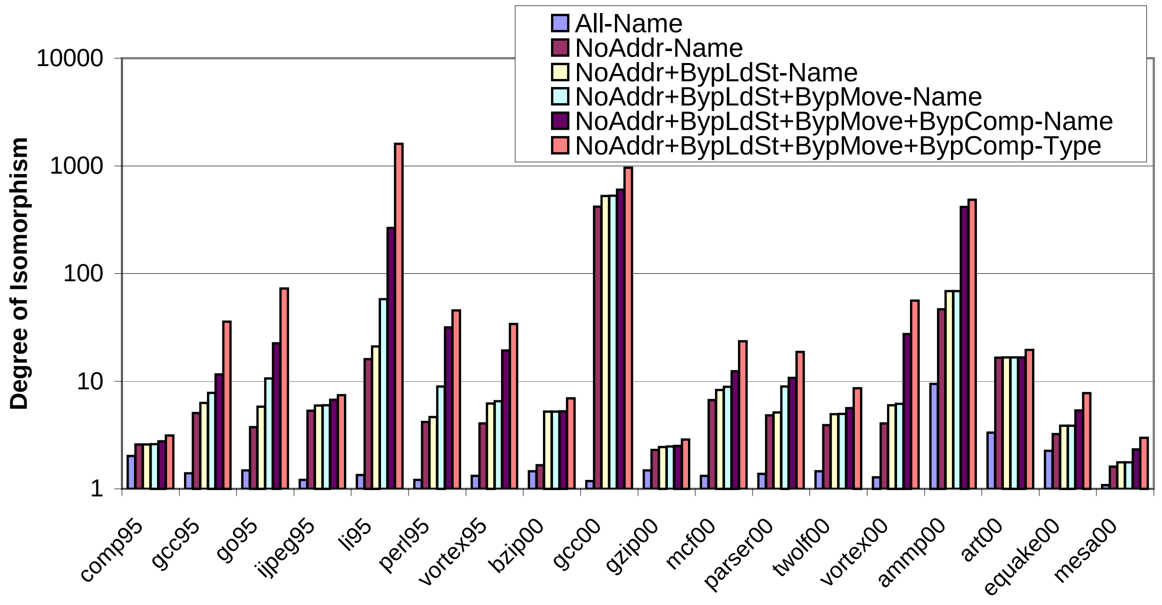


Figure 5: Isomorphism for various Combinations of Transformations

used. A subsequent section discusses program constructs and situations that influence isomorphic behavior.

Overall, the above observations demonstrate that the proposed transformations are conducive in increasing isomorphism. This also provides support to the claim that isomorphism is suppressed due to architectural semantics and that the proposed transformations are effective in converting non-isomorphic equality to isomorphic equality.

For the remaining paper we focus on the configuration *NoAddr+ BypLdSt+ BypMove+ BypComp-Type*.

5.2 Isomorphism Run Time Behavior

Fig. 6 shows the run time behavior in terms of non–isomorphic instructions with increasing instruction count. These instructions are shown as a fraction of the total dynamic instruction count for each benchmark. The last point on each curve corresponds to the fraction of dynamic instructions that defined unique components for a given benchmark run.

The difference between 100% and the value of the last point in Fig. 6, provides the fraction of instructions that were isomorphic for each benchmark. The data show that, depending on the benchmark, 65% (*gzip00*) to 99.9% (*li95*) of the dynamic instructions are isomorphic.

It is noteworthy that all programs produce throughout their execution new unique components. This indicates that typically programs form new computational patterns all during their execution. Virtually all benchmarks, at the beginning of their execution create a large number of unique isomorphic components. Specifically, during the first million instructions the unique components range from 29566 for *gcc00* to 615378 for *gzip00*. After, this "warmup" phase, most programs get into a steady state of reusing previously created components and producing roughly the same amount of new unique components. Notable exceptions are *ijpeg95* and *bzip00*. For *ijpeg95* the number of new components created fluctuates. Whereas for *bzip00* the simulated region consists

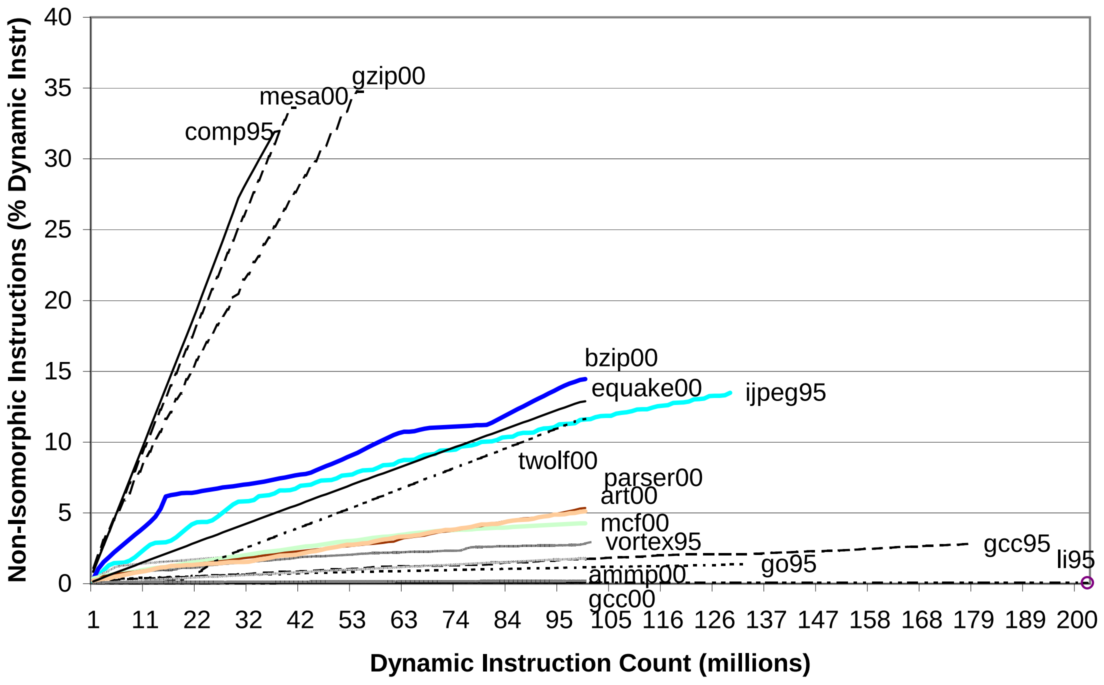


Figure 6: Run Time Behavior

of two distinct phases each with steady behavior. For three benchmarks, *ammp00*, gcc00 and *li95* the number of non–isomorphic instructions increases at a small rate not visible in the scale of this graph.

The isomorphic behavior was investigated for long simulations of a subset of SPEC2000 benchmarks to confirm the above observations. These data (not shown here) revealed that for longer runs more phase changes can be observed. However, for most benchmarks the isomorphism gets into a steady state for long sequence of instructions. More importantly, the isomorphism of longer runs is comparable or higher than shorter runs.

5.3 Structure of Components

Fig. 7 shows the distribution of dynamic instructions according to their component depth. For almost all benchmarks the components depths are smaller than 16384 nodes. This indicates that most programs do not contain constructs that produce long recurrences and that the transformations employed are effective in reducing components size. However, this is not true for *parser00*, *comp95*, *art00* and *gzip00*. These benchmarks have significant number of instructions with components larger than a million nodes. And each node on such a chain defines a unique component. Therefore, one might have expected a correlation between large depth and low isomorphism. Examination of the results in Fig. 5 and 7 reveals that such correlation does not exist. For example, *art00* with a lot of large components has higher isomorphism than *ijpeg95* that has shorter components. Additional criteria about the structure of components may be needed to establish correlation of depth with the observed degree of isomorphism.

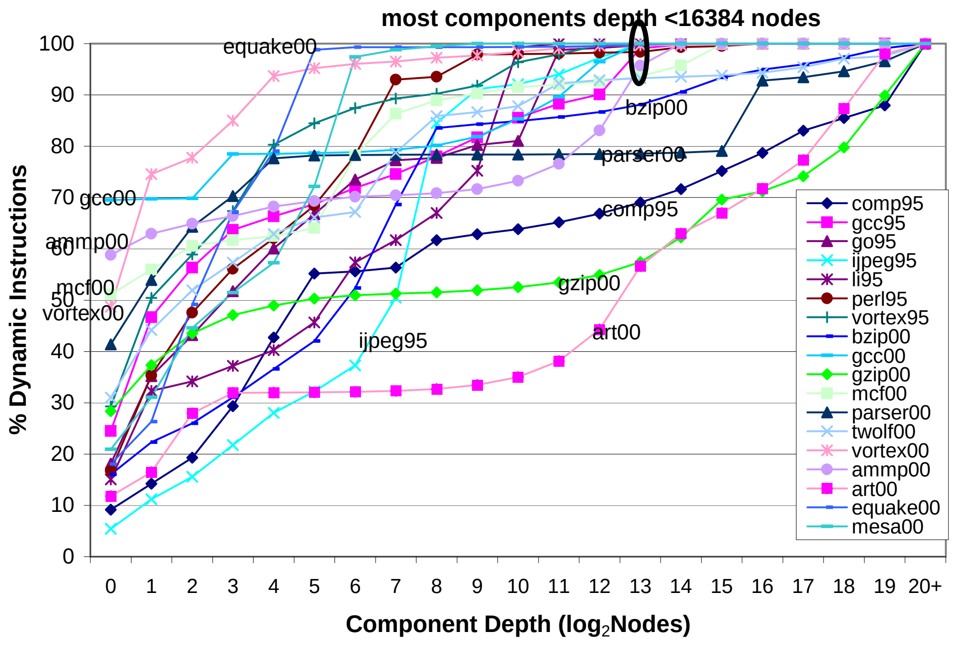


Figure 7: Distribution of Dynamic Instructions according to Component Depth

One other result from Fig. 7, is that some benchmarks have a large fraction of instructions with small depths. Specifically, for three benchmarks - *gcc00*, *ammp00*, and *mcf00* - more than 50% of the instructions have components with one node the instruction itself (this corresponds to point 0 in the graph). This behavior may be caused by at least one or combination of the following: transformations, program structure, and simulation methodology.

Program structure can produce short components if a program is reading frequently input data. Also transformations can cause this behavior because they can convert a load-use dependence chain into two independent smaller chains. For example, a load-use dependence can become parallel when the address dependence is ignored and the load is bypassed. However, further investigation revealed that simulation methodology is the main cause of this behavior. In particular the simulation for SPEC2000 benchmarks started after a region was skipped. As a result the previous state was represented with value nodes not the computation that produced it. This means that at each use of previous state would lead to a component of depth one. Note that SPEC95 complete simulations have small amounts of short components.

To gain more insight about the relation and structure of components, Fig. 8 shows the distribution of dynamic instructions according to the classification in Section 3.1. The results show that SPEC95 and SPEC2000 benchmarks have similar fraction of *source* instructions, 1–10%, but SPEC2000 benchmarks have significant more *source-maximal* instructions. Recall, these are nodes that have depth one and not used by any other instruction. Therefore, as argued above, this difference between SPEC95 and SPEC2000 can be attributed to simulation methodology.

Another observation, is that the instructions in the *middle* of components are smaller than the *maximal*. This is because for many components the middle computation is similar. This is supported by analysis that revealed very few components to be *extractable*, i.e. usually *maximal* components share middle nodes with other *maximal* components.

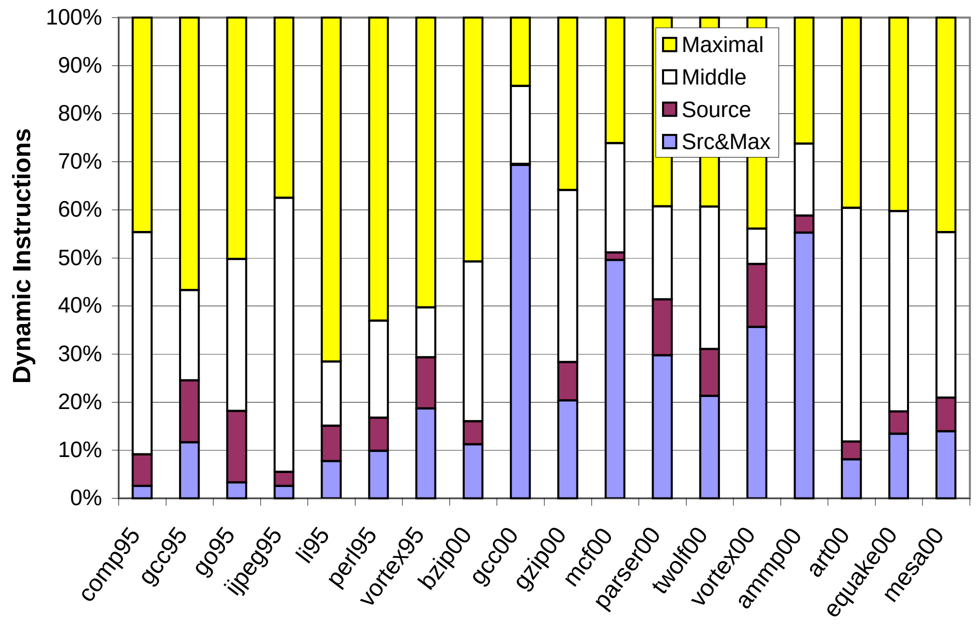


Figure 8: Distribution of Dynamic Instructions according to Component Structure

Component Locality: To investigate whether components for a given benchmark exhibit locality, Fig. 9 shows the minimum fraction of non-isomorphic instructions required to cover a fraction of dynamic instructions. The 10–90 rule appears to hold for most benchmarks. It is noteworthy that the first 1% is sufficient to cover in almost all cases more than 50% of execution. The data reveal a strong correlation between component locality and degree of isomorphism of a benchmark. The better the locality the higher the isomorphism. For example the benchmarks that are slower to reach the 90% mark are the benchmarks with the lowest isomorphism: *comp95*, *gzip00*, *mesa00*, *bzip00*, *ijpeg00* and *equake00*. The exception to the above is *ammp00* that requires a lot, 30%, of unique components to cover 90% of the execution, however is the fastest to reach very close to 100% coverage which explains its high degree of isomorphism.

5.4 Benchmark Analysis

Below we provide some analysis for the causes of isomorphic and non-isomorphic behavior exhibited by some of the benchmarks.

The main reason for the low isomorphism exhibited by *compress95* is a very long dependence chain that is created for the construction of a random text file to be compressed. Specifically, the leaf function *ran2* contains a sequence of interdependent instructions and no loops. The function is called 149999 times and each invocation carries a dependence from the previous call through a global register. Each node in this long dependence chain, depth of 1.8 million, defines a unique component. After this long dependence is created, 149999 of its nodes correspond to the data to be compressed. Each of these nodes is read once (minimal fanout) and processed by computation that forms a thin dependence chain. This naturally produces more unique components and no isomorphism. It is unclear how such a program structure can be represented in a more concise form.

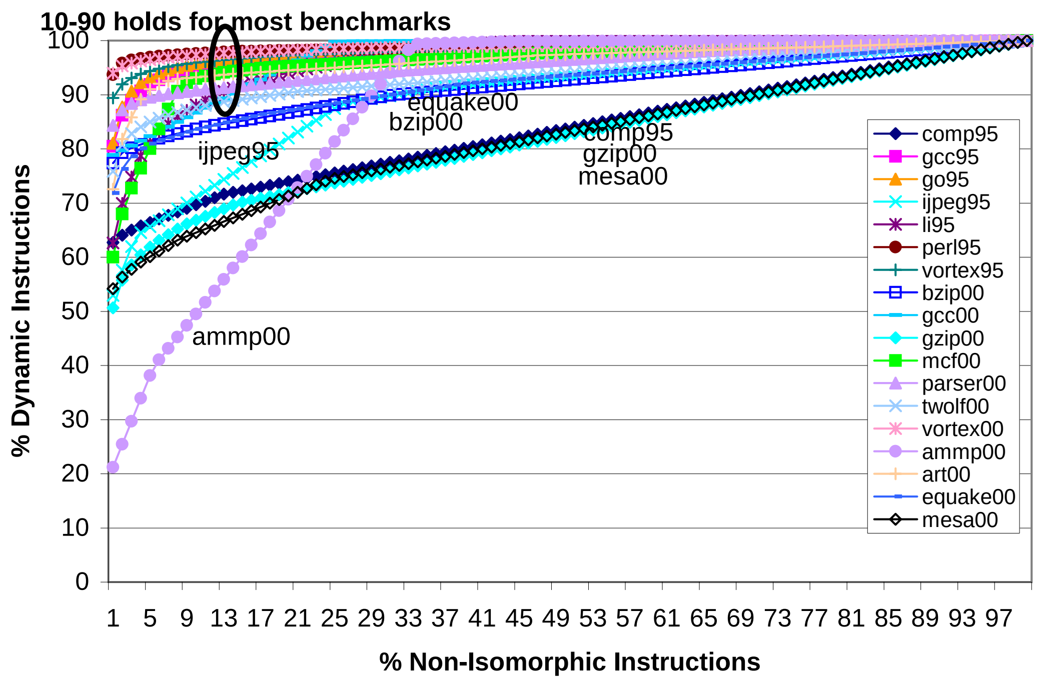


Figure 9: Component Locality

One of the main causes for the low isomorphism of *gzip00* is also a deep dependence chain with minimal fanout. The dependence is created in the leaf function *updcrc* that produces a code to check for errors during transmissions of the file to be compressed. This code is checked against a code in the file. To create this code a running hash of all data in the file is computed. And this leads to a deep dependence chain of depth larger than 1.2 million. Is unclear how such structure can be represented more concisely.

Note the dependence chains of million depth are significant because the simulations are typically no more than 100 million instructions long. In fact, for *compress00* and *gzip00* the simulation are run only for 37 and 54 million instructions respectively.

The above observations may be interpreted as suggesting that deep dependence chains imply low degree of isomorphism. However, benchmark *art00* with dependence-chains-depth in excess of a million shows high degree of isomorphism (see Fig. 5 and 7). Also the converse may not be true, for benchmark *ijpeg95* the deepest dependence chain was less than 16000 nodes, but *ijpeg95* did not exhibit high isomorphism.

The reason for the behavior of *ijpeg95* is programming constructs that can produce more unique components than the deepest dependence chain of the construct. In *ijpeg95* this is caused by a doubly nested loop with roughly the following structure:

```
for(i=x;i<x+524288;i+=64){ ...
   for(j=i;j<i+64;j+=2){ ...
   } ...
}
```

The depth of the dependence chain that updates the value of the outer induction variable *i* is 524288/64, i.e. 8192. The largest depth of the dependence to update the inner induction variable *j* is 8192+32, i.e. 8224. However, the above loop structure will produce 262144 unique components

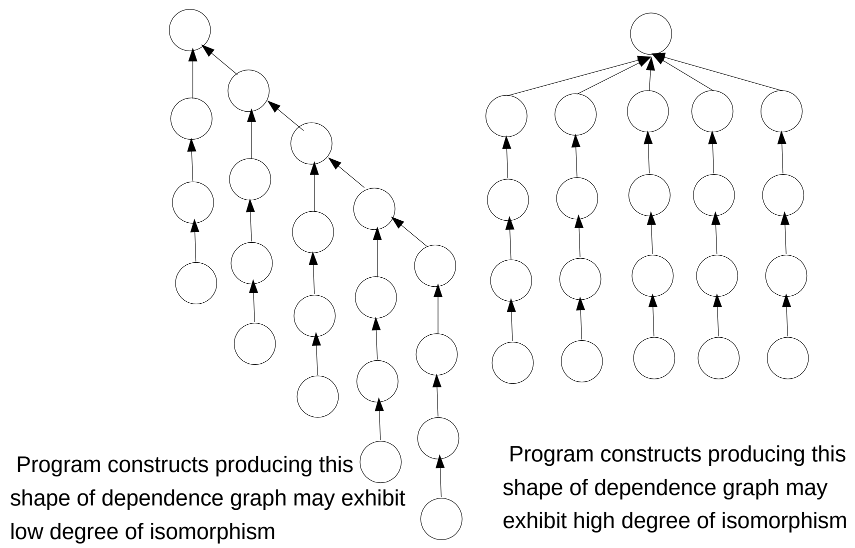

Program constructs producing this shape of dependence graph may exhibit low degree of isomorphism
Program constructs producing this shape of dependence graph may exhibit high degree of isomorphism

Figure 10: Program Structure and Isomorphism

one for each dynamic definition of *j*. The inner loop is contained in function *encode_one_block* and the outer loop in function *encode_mcu_huff*. This demonstrates that is possible to have large number of unique components while they are short and have same depth.

Art00, on the other hand, produces deep dependence chains but exhibits high isomorphism. Investigation of its structure revealed that most of the instructions are due to function *train_match*. This functions contains double, triple and quadruple nested loops. The nesting structure leads to very deep dependence chains for the computation of some variables. However, only a small subset of the nodes in this deep dependence chain are used to compute other values. And each time a node is used, is read several times and leads to isomorphism.

The above suggest that programs exhibit low isomorphism when there is a combination of a construct that produces a lot of unique components with small fanout that is thin. Conversely, programs that have nodes with high fanout are more likely to exhibit isomorphism. This two scenarios are shown pictorially in Fig. 10.

The behavior of *gcc00* was also investigated due to the very high amount of isomorphism it exhibited. It was determined that most of execution was spend in function *_wordcopy_fwd_aligned*. This function was called 837 times and each time a loop that iterates 5147 times was executed. This loop was unrolled 8 times and its body consisted of 20 instructions. Further investigation revealed that *_wordcopy_fwd_aligned* was used by the function *thread_jumps* for initializing a table with virtual registers. This function is used to adjust two branches when the second depends on the first and the two branches test the same condition. A web search revealed that others have observed that for some architectures the function thread_jumps is a *nop* and this may be caused by a performance bug. This may indicate that doing isomorphic analysis may be useful for detecting algorithmic or logical errors in programs. The above also suggests an execution approach where expensive initialization is not performed until is required.
Other benchmarks were analyzed and the following were found to be typical causes of (non)isomorphism:

- repeated traversals of identical or slightly modified link lists (isomorphism, in *li95* and *ammp00*),
- event counters that produce long dependence chains but with no fanout (non-isomorphism),
- repeated loading of a global variable due to register pressure (isomorphism), and
- repeated save and restoring in non-leaf functions when the function is effectively a leaf (isomorphism).

6. Conclusions and Future Work

This work identifies a new program runtime property: instruction–isomorphism. The paper provides a basic classification of different types of isomorphic behavior and introduces a number of transformation for converting non-isomorphism to isomorphism.

Instruction–Isomorphism is investigated empirically using SPEC benchmarks. The data show that program isomorphism is infrequent when considering all dependences. However, when using a combination of transformations the data show that, depending on the benchmark and dataset, 65 to 99.9% of the dynamic instructions are isomorphic. This suggests that instruction–isomorphism is suppressed due to architectural semantics and programming conventions that introduce a lot of "overhead" instructions and dependences. The proposed transformations lead to concise dependence graphs that rarely have depth greater than 16384 instructions. The results show that a small fraction, about 10%, of the computational patterns that occur during execution are needed to cover 90% of the dynamic instructions.

Individual benchmark analysis reveals: (a) the existence of program constructs necessary for execution that produce non-isomorphism, (b) program constructs that produce short dependence chains and can create more new components than their maximum dependence chain depth, and (c) that the higher the fanout from dynamic instructions the higher the isomorphism.

The presence of instruction–isomorphism in programs represents a call for several directions future work. One such direction is to establish methods so that current and next generation processors can benefit from isomorphic behavior: (a) performing "early" computation by detecting isomorphism in the dynamic dependence structure, (b) prediction schemes that consider information from the component of an instruction. Value prediction and branch prediction schemes that effectively rely on component information have already been proposed [28, 29, 30], and (c) establish how much of the history information used by current predictors influence the construction of a component and what is the significance of this relation.

Other directions of research are: (a) develop practical methods for (approximate) detection of isomorphism in programs, (b) use isomorphism as a criterion for deciding on the similarity/differences across programs to select benchmarks, and (c) measure isomorphism to assess the quality or detect errors in compiled code.

Acknowledgements

The author wish to thank Andromachi Katsonouri and Pedro Trancoso for their assistance in preparing this manuscript. We also thank the anonymous reviewers for their comments.

References

[1] J. E. Smith, “A Study of Branch Prediction Strategies,” in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[2] J. Lee and A. J. Smith, “Branch Prediction Strategies and Branch Target Buffer Design,” *IEEE Computer*, vol. 17, pp. 6–22, January 1984.

[3] J. L. Baer and T. F. Chen, “An Effective on-chip Preloading Scheme to Reduce Data Access Penalty,” in *Proceedings of Supercomputing’91*, pp. 176–186, November 1991.

[4] 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*, pp. 138–147, October 1996.

[5] F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction,” Tech. Rep. EE-TR-96-1080, Technion, November 1996.

[6] 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*, pp. 181–193, June 1997.

[7] 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*, pp. 35–45, October 1998.

[8] J. Huang and D. J. Lilja, “Exploiting Basic Block Value Locality with Block Reuse,” in *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 106–114, January 1999.

[9] Y. Sazeides and J. E. Smith, “Modeling Program Predictability,” in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 73–84, June 1998.

[10] A. Sodani and G. S. Sohi, “Dynamic Instruction Reuse,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194–205, June 1997.

[11] A. Sodani, “Dynamic Instruction Reuse,” PhD Thesis, University of Wisconsin - Madison 2000.

[12] D. A. Connors and W. mei Hwu, “Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results,” in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 158–169, November 1999.

[13] J. Huang and D. J. Lilja, “Extending Value Reuse to Basic Blocks with Compiler Support,” *IEEE Transactions on Computers*, vol. 49, no. 4, pp. 331–347, 2000.

[14] C. Molina, A. Gonzalez, and J. Tubella, “Dynamic removal of redundant computations,” in *Proceedings of the 13th International Conference on Supercomputing*, pp. 474–481, June 1999.

[15] S. Onder and R. Gupta, "Load and Store Reuse Using Register File Contents," in *Proceedings of the 15th International Conference on Supercomputing*, pp. 289–302, June 2001.

[16] V. Petric, A. Bracy, and A. Roth, "Three Extensions to Register Integration," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 37–47, November 2002.

[17] Y. Sazeides, "Instruction-Isomorphism in Program Execution," in *Proceedings of the 1st Annual Value Prediction Workshop (affiliated with ISCA-30)*, pp. 47–54, June 2003.

[18] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *8th International Symposium on Static Analysis*, pp. 40–56, July 2001.

[19] J. R. Larus and S. Chandra, "Using Tracing and Dynamic Slicing to Tune Compilers," Tech. Rep. CS-TR-93-1174, University of Wisconsin-Madison, August 1993.

[20] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[21] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[22] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 246–256, June 1990.

[23] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[25] S. Sechrest, C.-C. Lee, and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 22–32, May 1996.

[26] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–258, December 1997.

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

[28] 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*, pp. 107–117, September 2001.

[29] Y. Sazeides, "Dependence Based Value Prediction," Tech. Rep. CS-TR-02-00, University of Cyprus, February 2002.

[30] 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*, pp. 314–323, June 2003.