Characterizing Coarse-Grained Reuse of Computation

S. Subramanya Sastry, Rastislav Bodik, James E. Smith
sastry@cs.wisc.edu, bodik@cs.wisc.edu, jes@ece.wisc.edu
University of Wisconsin, Madison

Abstract

Value locality is the phenomenon that a small number of values occur repeatedly in the same register or memory location. Non-speculative *reuse of computation* [21] is one of the methods that has been proposed to exploit value locality. However, reuse becomes profitable only when multiple instructions are reused simultaneously. Identifying suitable chains of reusable instructions requires a global view of the program and is therefore difficult to be accomplished with hardware alone.

This paper investigates the properties of reuse in the context of a dynamic optimization setting. We focus on characterizing the available computation reuse in programs at coarse granularities, and in determining the relative applicability of specialization and memoization, two commonly used techniques for exploiting coarse-grained reuse.

Our study suggests that a reuse technique based on an online optimization system is feasible due to the following reasons. First, programs contain many large regions: on average, regions of 16 or more dynamic instructions represent 54% of all reuse, or 26% of all dynamic instructions. Second, large regions are stable over time and hence can likely be identified cost-effectively with a dynamic optimizer. Third, we observed that many reuse opportunities cannot be exploited with *memoization*, the currently used reuse technique, but must be supplemented with *specialization*, a technique that requires run-time code generation.

1 Introduction

Recently, a number of studies have demonstrated that programs exhibit significant *value locality*, the phenomenon that a small number of values occur repeatedly in the same register or memory location [6, 8, 11, 17, 21]. Microarchitectural techniques exploiting value locality follow one of two paradigms: *value prediction* or *computation reuse*. While prediction-based techniques improve performance by breaking data dependences [11, 12, 18], reuse-based techniques improve performance by reducing computation latency ([3, 8, 21]).

This paper focuses on the reuse paradigm. Typically, computation reuse works by remembering in a *reuse table* the inputs and outputs of a computation. The computation is either a single instruction or a set of instructions. When the computation occurs again with the same inputs, the previously computed results are obtained from a reuse table and the reusable instructions are bypassed.

Sodani and Sohi observed that up to 90% of dynamic instructions can be removed through reuse performed at the level of individual instructions [21]. The primary obstacle to exploiting this reuse potential is that to gain benefit, a chain of *multiple* instructions must be reused simultaneously; reusing a single instruction typically does not amortize the latency of looking up the reuse table. This identification of the set of instructions to be reused, *region identification*, constitutes one of the principal problems of reuse.

A number of hardware techniques have been proposed to identify and exploit coarse-grained reuse: linking data-dependent instructions in a h/w table [21], detecting reuse at the granularity of basic blocks [8], and trace-level reuse [6]. More recently, a hybrid reuse technique using a combination of software and hardware was proposed by Connors and Hwu [3] wherein a compiler identifies reuse "regions" by consulting an *off-line* value profile. The hardware is then responsible for recording the execution instances of these regions and reusing them

In the compiler domain, it has been known for a long time that opportunities exist to speed up programs by exploiting knowledge of fixed/invariant inputs. These software techniques to exploit knowledge of invariant inputs are typically based on partial evaluation [9], in which the program is *specialized* for the invariant/fixed inputs. These program specialization optimizations are applied to entire regions of code that operate on the fixed inputs. Temp [4], DyC [7], Data Specialization [10], tcc [16], code specialization using value profiles [15] are all software techniques for exploiting coarse-grained reuse.

All known techniques for exploiting coarse-grained reuse can be categorized as either a *memoization* or a *specialization* technique. Memoization is a technique based on looking up previous results in a reuse table [2]. Specialization involves optimizing the program by hardcoding the values produced by a reusable piece of code. Although various specialization techniques [4, 7, 15, 16] have been studied, available opportunities for specialization in general-purpose programs has not been studied before.

The goal of this paper is to investigate reuse using a trace-based measurement of its properties. We perform our study in the context of an online, dynamic optimizer. In particular, we want to answer the following questions: (**i**) What is the amount of reuse that comes from "large" regions of instructions? (Reuse is likely most useful when reuse exists at large granularities.) (**ii**) Are the large reuse regions "stable" over time? If yes, the region identified by a dynamic optimizer may be used for a long time, which amortizes the software cost of identifying it. (**iii**) How should reuse be implemented? What are the respective advantages of memoization and specialization for exploiting reuse?

The results of our study can be summarized as follows:

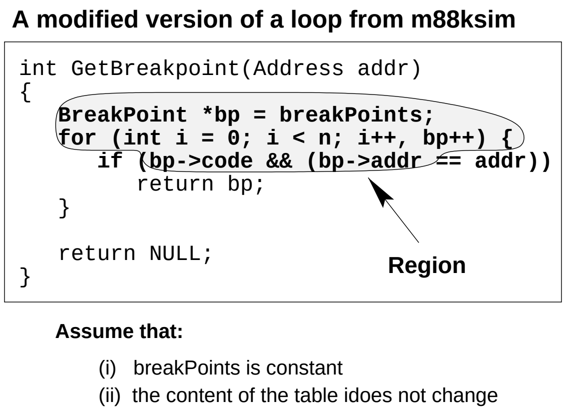
- *(i) Our benchmarks contain a significant level of reuse from large regions.* On average, 66% of all reuse (or 29% of dynamic instructions) comes from regions containing 8 or more dynamic instructions. Regions of sizes 16 or more represent only slightly less reuse, 54%, which is 26% of all dynamic instructions. On regions of size 8 or more, the cost of looking up the reuse table will likely be much smaller than the benefits of bypassing the reused instructions. This conclusion is confirmed by the results of Connors and Hwu who showed that reuse of large regions leads to significant performance improvements [3].
- *(ii) Reuse regions are highly stable over time.* We observed that, typically, the same region of instructions is reused under the same inputs tens of times (in a rather short program trace). This regularity of a region's shape suggests that, with a hardware value profiler, we can afford to identify regions via software at run-time, since the overheads are likely to be paid back.
- *(iii) Specialization should be used together with memoization.* Our results show that, for many benchmarks, specialization is more suitable than the currently used memoization, because (a) most of their regions reuse the same (single) value, hence lookup in the reuse table is not necessary; or (b) the shape of their regions requires run-time code generation, which is implicit in specialization.

The remainder of this paper is organized as follows. Section 2 discusses our notion of regions and presents the methodology we use to identify reusable regions. In Section 3, we provide data that characterizes reusable regions and draw conclusions from the data. In Section 4, we briefly discuss the characteristics of an online algorithm to build regions in a feedback-directed optimization setting. We finally summarize in Section 5 the lessons learned from this study.

2 Methodology

2.1 Region definition

We perform our study of reuse at the granularity of regions. Informally, we consider a *region* to be a set of *dependent* instructions that can be reused simultaneously, but they need not be contiguous. A region can consist of instructions from multiple basic blocks without necessarily containing all instructions of those basic blocks. A reuse region can also include multiple loop iterations and can span procedure calls. Our regions have unrestricted structure because our goal is to let the characteristics of reusable regions guide the development of reuse optimization techniques rather than the other way round.



A modified version of a loop from m88ksim

```
int GetBreakpoint(Address addr)
{
   BreakPoint *bp = breakPoints;
   for (int i = 0; i < n; i++, bp++) {
      if (bp->code && (bp->addr == addr))
         return bp;
   }

   return NULL;
}
```

Assume that:

(i) breakPoints is constant
(ii) the content of the table idoes not change

Figure 1: Example of a region

Figure 1 is an example of a reusable region. The figure shows a simplified version of the *ckbrkpts* function from *m88ksim*, a SpecINT95 benchmark. The loop searches through a breakpoint table for the breakpoint at `addr`. In order to create a reuse opportunity, let us assume that the content of the breakpoint table does not change after the table is initialized. Under this assumption, the value of `addr` is the only input value that changes across multiple calls to this function. Consequently, all instructions in the loop except for `bp->addr == addr` are reusable. The corresponding reuse region is shown in the figure. The region is composed of a dynamically non-contiguous sequence of instructions (although the reused instructions are contiguous in the static program, in the dynamic instruction stream the reused instructions are interrupted with the non-reusable "==" instructions).

More formally, a region is defined to be any arbitrary *connected* subgraph[1] of the *dynamic program dependence graph* (DPDG) which is constructed from a program trace.

[1]We require that there be no path in the graph from the output of a region to its input. This constraint ensures that the input of a region does not depend on its output.

The nodes of the DPDG consist of all the dynamic instruction instances in the trace. The edges of the graph consist of all register data dependences edges, all store-load memory dependence edges, and all control dependence edges.

Given this definition of a region, region reuse is then defined in terms of "equality" of these subgraphs, i.e. two regions are reusable if they have "identical" subgraphs.

2.2 Region-level Reuse

Let us further examine the notion of reusable regions by drawing an analogy with the well-understood notion of instruction reuse. Instruction I_2 is considered to be a reusable instance of an *earlier* instruction I_1 if both I_1 and I_2 are the same *static* instruction, take the same inputs and produce the same outputs. Region reuse is then a straightforward generalization of instruction reuse. Region R_2 is considered a reusable instance of an earlier region R_1 if both R_1 and R_2 perform *identical* operations, accept the same inputs and produce the same outputs. An additional constraint is that R_1 and R_2 must also be disjoint. Two regions are considered to perform *identical* operations if they have an identical set of instructions and have an identical set of dependences, i.e. their corresponding dependence subgraphs are isomorphic. Potentially, a large set of regions can all have "identical" structures and have identical inputs and outputs. It is these sets of reusable regions we are most interested in, because they are good prospects for specialization and memoization.

Given these notions, an algorithm for detecting reuse works by identifying identical subgraphs in the DPDG.

2.3 Identifying Reusable Regions

Our study simulates benchmark programs, builds the dynamic dependence graph, and partitions them into reusable regions. During this partitioning, each dynamic instruction appears in at most one region. Note that even though a dynamic instruction is part of at most one region, the corresponding static instruction can map onto multiple regions.

Since we are interested in an online setting, there is an initial cost involved in building reuse regions. Besides this, both specialization and memoization have other associated optimization overheads. Specialization has only an **initial (high)** optimization cost. This cost can be amortized only if the same region is encountered repeatedly. Therefore, we require a high repeat-rate for regions that are specialized. Memoization, on the other hand, involves an initial set-up cost which tends to be much lower. Thus, a lower repeat-rate is sufficient for memoization. But, memoization also incurs overheads in the form of table look-ups and table space **each time** the region is encountered. Therefore, one large region will yield a greater benefit than multiple smaller regions. Hence, for memoization, large regions are desirable.

For a specialization-based optimization, a high repeat rate is more important than large regions (even though large regions are likely more beneficial than smaller regions). But, for a given trace length, a region of size N will have a higher repeat rate than a larger enclosing region of size $N + 1$. Hence, larger regions tend to have a lower repeat rate than smaller regions. Therefore, we have two conflicting requirements in our study since we are interested in both memoization as well as specialization.

To strike a compromise, we first require that reuse regions are replicated at least τ times – the *reuse threshold*. We will collect data for a range of thresholds, but for a particular optimization technique, this threshold should be high enough for the benefits of optimization to offset the cost of identifying the region, any code re-structuring, and/or hardware table setup. Then, subject to the threshold constraint, we maximize region size. We now have an optimization problem. Subject to the threshold constraint, we would like to find the smallest set of regions that maximizes instruction coverage (i.e. the number of instructions that are part of a region). In the next section, we will consider an algorithm for finding such *maximal* reuse regions.

2.4 Region Building Algorithm

We very briefly describe how our algorithm identifies regions. Conceptually, at the simplest, a dynamic instruction x forms a one-instruction region. By linking up x with all its earlier reusable instances (with same inputs and output), the algorithm builds a reuse chain for x, which also serves as a chain of reusable one-instruction regions. Next, this one-instruction region containing x is expanded into a two-instruction region R_2, by adding y, a dependent instruction of x. The algorithm synthesizes the reuse chain for the 2-instruction region R_2 by "combining" individual nodes in the reuse chains of x and y. This process continues till the region cannot be expanded any further (i.e. the reuse chain for region R_k contains fewer than τ instances).

The algorithm is *greedy*. **(i)** It always grows a region as big as possible (while satisfying the reuse threshold), and **(ii)** if a region R can be grown by adding a new node x, it does so without exploring other alternatives. Hence, this algorithm can yield sub-optimal results in terms of instruction coverage. We followed this approach because the complexity of the problem (non-unique partitioning into regions) does not allow for an exhaustive search.

Sub-optimality can also result from other practical considerations of space and time constraints that arise in implementing the above algorithm. It is not possible to analyze an entire program trace T in one pass because the dependence graph for T and the reuse chains consume a lot of memory and analysis time. To tackle this, the algorithm breaks up T into smaller windows and analyzes the windows one at a time, and carries over some minimal summary information across windows. For the *go* benchmark, with a window size of 0.75M instructions, the instruction coverage increased from 11.7% to 16.8% over 12M instructions. With a window size of 1.5M instructions, the coverage increased from 11.8% to 17.6% over 12M instructions. This indicates two things. First, by increasing the length of the trace, greater number of regions are identified. Second, the loss in coverage by breaking up into windows is tolerable.

In spite of these drawbacks, our greedy algorithm does establish a *lower bound* on the numbers and sizes of regions that exist in programs.

For this study, we used the functional simulator from the Simplescalar toolset [5]. All analysis was performed by collecting a dynamic trace of instructions and constructing a dynamic program dependence graph from the trace. For the purposes of this study, we eliminate trivially reusable instructions (direct jump, direct call, and a code idiom that initializes addresses). We also eliminate fill-refill and save-restore code and connect producers and consumers. We also ignore data dependences on the stack pointer and return address register since these dependences artificially constrain reuse.[2] Overall, the transformations either eliminate or ignore roughly 5% to 15% of instructions.

2.5 Benchmarks

We conducted this study on a collection of SpecINT95 benchmarks, "micro-benchmarks", and Java programs. Table 2.5 lists the programs we used for our study, the simulation window size for each benchmark and the total number of instructions simulated. For all these programs, simulation was performed by skipping over the initialization phases of the programs. For the SpecInt95 benchmarks, we used the recommendations of Sherwood and Calder [19] in determining the starting point of our simulation. For the Java benchmarks, the starting points were determined empirically.

The choice of window size and trace length was determined by the amount of available memory. Even though we collect our data for short traces, we believe that our results are still meaningful, i.e. they characterize behavior in at least a subset of important hotspots. Except for *go* and *gcc*, all the other SpecINT95 benchmarks have few well-defined hotspots [14]. Hence, for all these benchmarks, the trace is likely to capture the execution of some hotspot. For example, the biggest hotspot of *perl* has 650 instructions [14]. Thus, the 0.5M trace for *perl* would capture at least 750 executions of any hotspot. Further, both *perl* and *m88ksim* are interpreters and hence we believe that the value profile seen in the short run reflects their behavior over longer runs of the program.

[2]If A calls B from different stack depths but with same parameters, the stack pointer will have different values each time and will break the reusable region at the call boundary.

The SpecInt95 benchmarks were compiled for the SimpleScalar ISA by *gcc* with optimization flags "-O3". The Java programs were compiled by the Strata compiler for the SimpleScalar ISA. Strata is a research Java bytecode to native ISA compiler (written in Java) that was developed as part of our research project [20]. This compiler does all the traditional local optimizations, eliminates null and array bound checks, and performs global register allocation. *interp* is an interpreter loop taken from [7]. The input to the interpreter is a factorial program. *printf* is a program where a `printf` like function prints data based on a format string. In this program, the formatting string is held constant, and the numeric data to be printed is changed within a loop.

3 Results

We use *Instruction coverage* as the primary metric in characterizing region reuse. A better metric is used in Section 3.5. *Instruction coverage* is the fraction of all dynamic instructions that are contained in reusable regions. These instructions can be removed if *each* region found by our study is reused. Although not each region will be removed in a realistic implementation (e.g. the region is too small), instruction coverage suitably estimates the overall effectiveness of region-level reuse as a performance enhancer.

3.1 Variation of Instruction Coverage with Reuse Threshold

First, we present the *instruction coverage* for all the benchmarks. Because reusing individual instructions rarely brings any benefit, single-instruction regions are not included in the statistics. Further, note that roughly 5%–15% of instructions are eliminated due to preprocessing of the program dependence graph (See Section 2). For some benchmark and reuse threshold combinations, the data points are missing because the simulations could not complete in available memory and/or in a reasonable amount of time.

In summary, our benchmarks contain a very high potential for region-based reuse. Most importantly, in most benchmarks, the reuse is not very sensitive to the threshold level. Even for a threshold as high as 100, between 10% and 60% of instructions can be removed, and across all benchmarks, on an average, about 35% of dynamic instructions can be eliminated.

We discussed in Section 2 that we study the reuse behavior across different thresholds to evaluate whether reusable regions are amenable to specialization, i.e. whether the

Benchmark	Comment	Window Size	Total Instrs	Input
go	SpecInt95	1.5M	12 M	5stone21.siz, 5stone21.in (Ref)
li	SpecInt95	1M	4 M	8-queens.lsp (Test)
ijpeg	SpecInt95	0.5M	3 M	Test
m88ksim	SpecInt95	0.2M	0.8 M	Ref input
gcc	SpecInt95	0.5M	0.5 M	cccp.i
perl	SpecInt95	0.5M	0.5 M	primes.pl, primes.in
raytrace	SpecJVM98	0.5M	1.5 M	Speed 100
strata	Java	0.5M	2 M	Compiles a Java class file
interp	Micro-Benchmark	0.1M	0.3 M	Factorial 70
printf	Micro-Benchmark	0.1M	0.3 M	Prints 1 through N

Table 1: Benchmarks used in this study

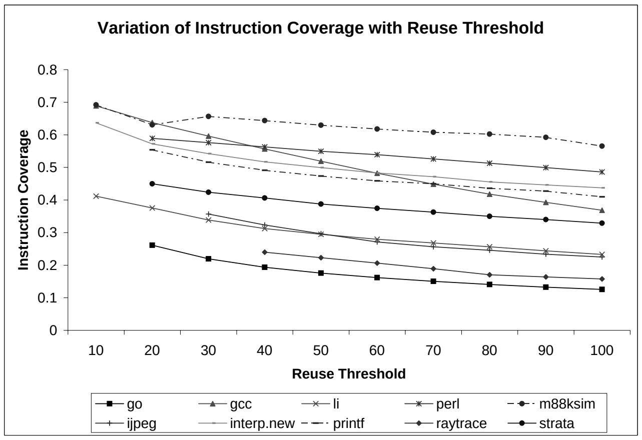


Figure 2: Variation of instruction coverage with Reuse threshold

benefits would offset the overheads. Our results indicate that there is potential for specialization within some of these benchmarks.

Figure 2 shows the variation of instruction coverage with the reuse threshold. The behavior differs across benchmarks, but as should be expected, there is a uniform reduction in instruction coverage as the reuse threshold is increased. For *go* and *gcc*, the coverage at a threshold of 100 is half of what it is at a threshold of 10 because both these benchmarks are known to execute a large number of program paths [1] and their hotspots do not cover large portions of their execution [14]. Thus, even though there exist reusable regions, the code containing them is not executed often enough to bring the reuse over the threshold.

For benchmarks like *m88ksim* and *interp*, the reduction in instruction coverage is much smaller because these benchmarks have well-defined hotspots with significant sources of reusability.

Caveats:
1. In this and the following graphs, different benchmarks have been simulated for different trace lengths. As indicated earlier, this choice was dictated by the memory and time requirements of the simulations. In Section 2.4, we noticed that instruction coverage increases (and likely stabilizes) with increasing trace lengths. Therefore, if all benchmarks were simulated to the same trace length of, say 10M instructions, the curves would likely look better.

2. For *gcc*, further experiments show that the behavior differs in different phases of the program. In the parsing and output phases of *gcc*, the instruction coverage is of the order of 20% unlike the *reload* phase shown in the result graphs. Nevertheless, the results indicate that *gcc* has phases where it exhibits high reusability.

3.2 Variation of Instruction Coverage with Region Size

Even though a large number of instructions are covered by reusable regions, it is more useful to study how these instructions are distributed amongst regions of different sizes.

Figure 3 shows the distribution of instruction coverage across different region sizes for a reuse threshold of 50. The bottom curve shows how often a region was reused; because each region reuse requires a lookup, this curve roughly corresponds to the cost of performing the reuse. The top curve shows how many instructions were reused, which roughly corresponds to the benefit of reuse. The further apart these curves, the greater the benefit. Note that the figure shows the distribution as a fraction of reusable instructions. The actual instruction coverage can be read from Figure 2. The figure shows results for *go*, *gcc*, *m88ksim* and *strata*. These are representative of the behavior of the other benchmarks.

At one end of the spectrum, *go* has most of its instructions in small regions. Over 70% of the reusable instructions are in regions of size 7 or less. The behavior of *li* is similar, though not as bad as that of *go*. For these

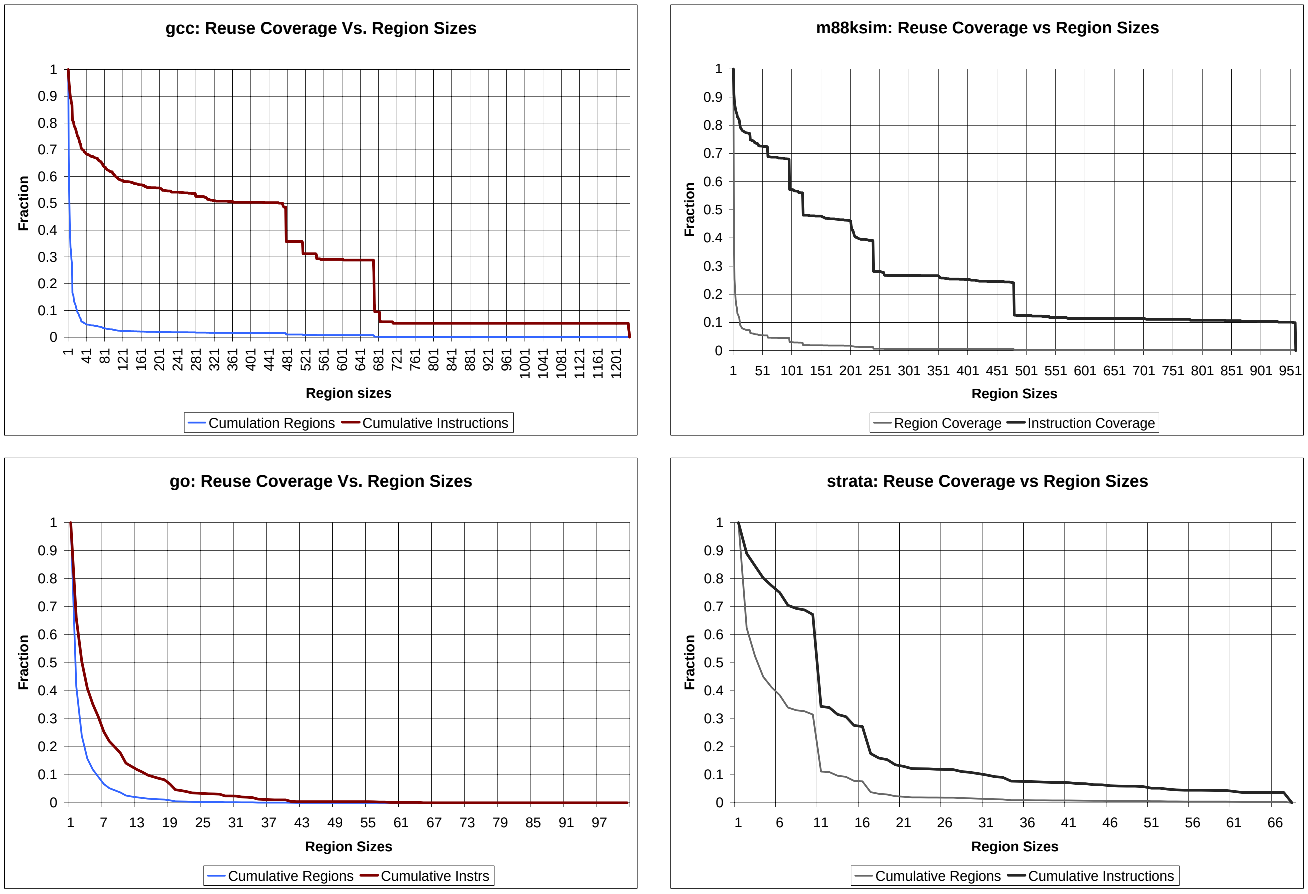


Figure 3: Cumulative instruction coverage across different region sizes for reuse threshold 50

benchmarks that have a predominance of small regions, one might require more fine-grained support in hardware (i.e., a faster lookup in the reuse table) to profitably exploit them. The *Strata* benchmark is slightly better. About 35% of the reusable instructions are in regions of size 12 or more. Thus, there is still reasonable reuse potential at bigger region sizes. *raytrace* and *ijpeg* fall in this category.

At the other end of the spectrum are *gcc*, *m88ksim*, *perl*, *interp*, and *printf*. For *gcc*, over 70% of the instructions covered are in regions of size 40 or more. Further, as the graphs show, there are certain well-defined jumps in the instruction coverage. These correspond to large regions that occur repeatedly. Likewise, with *m88ksim*, over 72% of instructions covered are in regions of size 50 or more. As with *gcc*, there are well-defined jumps in the instruction coverage indicating the presence of large regions that occur repeatedly. These results indicate that there is a lot of reuse potential in many programs at large granularities.

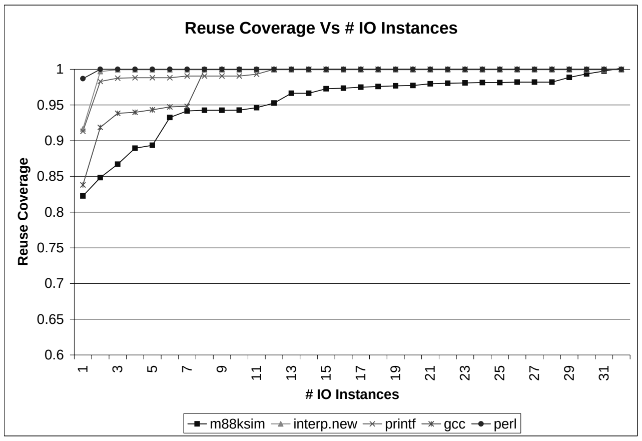
3.3 Variance of the input vectors in a region

We now answer the question of how many different values are reused in a region. Roughly, a region invoked with the same value always can be optimized with code specialization; a region with multiple values requires memoization.

The term *reuse region* we considered so far denoted a set of reused dynamic instructions together with their *particular* reused values (i.e., the input vector). In order to determine how many different input vectors appear in a given *static* region of code, we need to collapse the reuse regions according to their shape, thus discovering how many values appeared in the same region of dynamic instructions. More precisely, we collapse regions that are isomorphic (but may have different values).

Figure 4 shows the cumulative instruction coverage across regions with identical shape ("identical" regions) but different number of unique input vectors, called IO instances. Reuse on the left-most side of each curve can be removed with specialization; any additional reuse requires memoization. The first graph in the figure shows this distribution for *m88ksim, perl, interp, printf, gcc*. For all these benchmarks, over 80% of instructions are in "identical" regions that has exactly one input vector. Thus, these programs are very amenable to specialization. Except for *m88ksim*, the rest have almost all reusable instructions in code-equivalent regions with at most 8 unique input vectors.

The second graph in the figure shows the distribution for



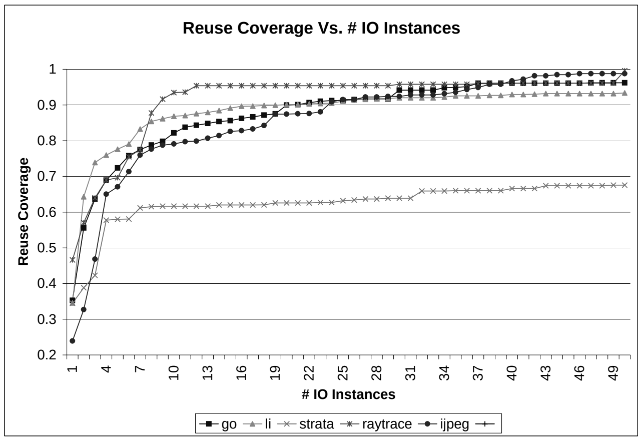


Figure 4: Distribution of input vectors (in the graph, called IO instances)

go, *li*, *ijpeg*, *raytrace*, and *strata*. There is a sharp difference with the first graph: only 25% - 45% of reuse comes from regions with one input vector. Due to the much higher degree of variation, memoization is likely to be more suitable than specialization.

An interesting observation is that except for *gcc*, the first group of programs on the left are interpreters. Our measurement thus empirically confirms the accepted wisdom that code specialization works particularly well for interpreters (because they operate on a rarely-modified array of instructions). Although *gcc* is not an interpreter, in the window of simulation, it probably operated on data that did not change.

The important conclusion is that both memoization and specialization are needed to exploit the available reuse.[3] This also indicates that an optimization framework that employs both memoization and specialization is likely to perform better than either one individually.

3.4 Load Behavior

If a region has internal loads, these loads act as *implicit* inputs to the region because stores can potentially change the value that is loaded across multiple invocations of the region. In the presence of loads, reuse implementations can either assume they always load the same value and somehow guarantee this invariance (via software or hardware) or they can restructure regions so that loads are never inside a region, at the cost of more lookups. In order to determine an appropriate strategy, we classify regions based on their load behavior as follows.

- If a region has no loads and no stores, it is a pure computation region without any side effects. These regions require no special support for memory protection.
- If a region has no loads but has stores, then stores must be performed to maintain correct program state. This region classification includes regions that have side-effects on the memory.
- Given a memory address, if a load always returns the same value, then it is considered a *static* load. Note that we only require value equality on memory locations, not on program PC. Thus, a load instruction can load from multiple memory locations, and for each of those addresses, the load can either be *static* or not. If a region has only static loads, it is classified as a *Static* region.
- A load that is not static is termed a *dynamic* load. That is, this load returns multiple values from the same memory location. If a region has at least one load that is static and at least one load that is dynamic, it is classified as a *Static+Dynamic* region.
- If a region has no static loads, then it is classified as a *Dynamic* region.

Figure 5 shows the instruction coverage across the different load classes. All benchmarks have non-trivial number of instructions in regions that are side-effect free. For *go* and *ijpeg*, this accounts for more than half the instruction coverage.

Static Loads: The significant result from Figure 5 is the predominance of instructions that are present in regions that have static loads (Static and Static+Dynamic regions). For all benchmarks, at least half the instruction coverage across all benchmarks comes from the Static and Static+Dynamic classes. This indicates that it is very important to have some hardware or software support for static loads. When such support is present, the reuse technique can assume the invariance of these loads and elim-

[3]Note that while memoization can sometimes be used where specialization is applicable, memoization cannot fully subsume specialization. For instance, the typical interpreter can be specialized [7] but not memoized.

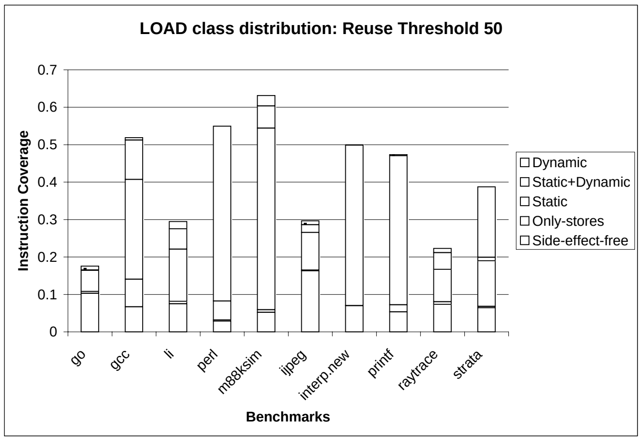


Figure 5: Region classification based on load behavior

inate them. To guarantee correctness, this would require some kind of detection scheme to detect violations of the assumptions.

Dynamic Loads : Studying Figure 5 further, we find that for *perl* and *printf*, almost the entire instruction coverage comes from regions that have dynamic (and static) loads. Smart region building techniques might be able to duplicate code such that within each copy, a load always return the same value. A more straightforward solution would be to break these regions into smaller regions and move the loads outside the region. Our results indicate that breaking regions to eliminate dynamic loads does not significantly affect region reuse and also makes static loads more prominent.

3.5 Regions: Contiguous or Non-contiguous?

In the absence of an actual implementation, we use the following equation to estimate expected benefits for the purpose of evaluating two different region building algorithms. The equation assumes a processor with 1 CPI and tries to model an implementation similar to Connors and Hwu [3].

Estimated execution time: $E = N - x + c * k + m * \sum EM(R)$

Estimated speedup = N/E

N – # instructions simulated

x – # instructions removed (as part of regions)

k – # *dynamic* regions

c – Lookup cost (assumed 2 cycles)

m – Miss penalty if lookup in the memo table misses (assumed 8 cycles)

$EM(R)$ = Estimated misses for *static* region R = $(n_I - s + 1) - n_R$ where:

I is the first instruction of R.

n_I – the number of times I executes

n_R is the number of times it is reused as part of R.

s is the first execution instance that I is reused. Hence, for the first $s - 1$ instances, there is no lookup for I. After the s^{th} execution of I, it is assumed that there is a table lookup each time I executes.

Caveats: This equation:

- Does not account for the cost of identifying regions and the cost of program transformation (memoization or specialization).
- Assumes a perfect memo-table (no capacity or conflict misses) in computing miss-rate
- Does not account for the cost of the memo table in terms of the size and its impact on latency.
- Assumes a table lookup cost for specializable regions too whereas in reality, such a runtime cost is not always incurred.
- Assumes that all instructions can be eliminated.
- Does not account for intelligence in the region-building algorithm that can eliminate more than the program analysis can identify (basically because of the sub-optimality of the region building algorithm)

For a region to be reused, its input vector is looked up in a memo table. Every lookup incurs the cost of lookup in a memoization table. On a hit, all the instructions in the region are bypassed which accrue as profit. On a miss, the processor has to fetch instructions from the actual region and this incurs a branch misprediction penalty. While this penalty is incurred on every lookup miss in the implementation of Connors and Hwu [3], this need not be the case.

We use this analytical model to evaluate region building algorithms rather than as an estimate of expected speedup from the optimizations. Using this model, we evaluate expected benefits from two region-building algorithms, one that builds contiguous regions and another that builds non-contiguous regions. Table 2 shows three sets of results for two different region building algorithms. For this evaluation, only regions of size 4 and greater are considered. Also, only static loads are included in regions and stores are not part of regions. These constraints attempt to mimic the model assumed by Connors and Hwu [3]. The first set of results is the instruction coverage as a fraction of total executed instructions. The second set of results is the fraction of reusable instructions that are in profitable regions. A **static** reusable region is considered *profitable* if the lookup and miss-rate overheads are smaller than the reuse benefit. The final set of results is the estimated speedups.

The results show that the estimated speedup has a positive correlation with the fraction of instruction coverage

Benchmark	*perl*	*m88ksim*	*gcc*	*interp*	*printf*	*go*	*li*	*ijpeg*	*raytrace*	*strata*
Trace Length	1.2M	1M	1.2M	1M	0.3M	20M	6M	3M	1.5M	3M
Instruction Coverage (as a fraction of total instructions)										
Non-contiguous	0.47	0.41	0.39	0.61	0.36	0.09	0.20	0.21	0.09	0.09
Contiguous	0.48	0.31	0.37	0.66	0.40	0.09	0.19	0.17	0.11	0.10
Fraction of reusable instructions that is profitable										
Non-contiguous	0.98	0.96	0.81	0.96	0.96	0.32	0.64	0.49	0.78	0.50
Contiguous	0.90	0.95	0.86	0.97	0.95	0.33	0.66	0.42	0.60	0.69
Estimated Speedup										
Non-contiguous	1.77	1.40	1.38	2.31	1.42	1.02	1.10	1.08	1.05	1.03
Contiguous	1.44	1.25	1.24	2.24	1.36	1.01	1.09	1.05	1.03	1.03

Table 2: Estimated speedups for different benchmarks

that is profitable. Therefore, as this fraction increases, the estimated speedup can be expected to increase. In summary, the results indicate that there is significant reuse potential in programs. These results were obtained with very short simulation runs. The speedups over the entire run of the program will likely be higher because: (i) the overheads of identifying reusable regions would get amortized; and (ii) with a longer run, the fraction of profitable reuse is expected to increase. This should explain the low analytical speedups for half the benchmarks when compared to the results obtained by Connors and Hwu [3].

But, the comparison between contiguous and non-contiguous regions is more meaningful and useful. The results show that the instruction coverage is pretty similar for both the algorithms (except for *m88ksim*). The results also indicate that for some benchmarks, non-contiguous regions can provide much greater benefits than contiguous regions. For *gcc*, *perl*, *m88ksim*, the gap seems to be significant. For the other benchmarks, contiguous regions do as well as non-contiguous regions. Based on these two results, we therefore conclude that for the most part, reuse optimizations should be biased towards contiguous regions and should consider non-contiguous regions only when the expected benefits are much higher.

However, the speedup gaps between the two algorithms is likely to be higher than these results indicate because the speedup equation does not account for the cost of misses due to a finite-sized memo table. While not shown here, the instructions are distributed across a much greater number of regions for the contiguous algorithm when compared to the non-contiguous algorithm. The region ratio is 3X for *gcc*, 7X for *perl*, and 5X for *interp*. Therefore, for a finite-sized memo table, the miss rate is likely to be smaller if large non-contiguous regions are exploited by specialization.

We conclude that memoization can yield greater speedups when assisted by specialization because (i) While contiguous regions are amenable to memoization, some kind of specialization is necessary to exploit non-contiguous regions. For some cases, a simple code restructuring might suffice to convert a non-contiguous region into a contiguous region which can then be memoized. (ii) Even when a contiguous region can be memoized, if the region has exactly one input vector, it is more profitable to specialize the function because specializing contiguous regions is straightforward (at least as hard as memoization).

4 Identifying regions at runtime

The previous sections presented a trace-based study of region-based reuse. Reuse regions were identified by constructing dynamic traces of instructions, building the dependence graph and identifying isomorphic subgraphs in the graph. However, in a feedback-directed optimization setting, region identification will not be based on building dynamic traces. Such a process is memory and time-intensive.

In a runtime optimization setting, region building will be based on constructing a value profile of selected instructions. At this time, we do not have a working algorithm for online identification of regions. However, any such algorithm is likely going to be based on collecting a value profile. The value profile is used to identify instructions that produce a small set of values. Using data and control dependence analysis, a set of dependent instructions can be collected to be part of a region. Existing offline static techniques for building regions [3, 13, 15] are structured along these lines. All these implementations perform region-building statically using a separate profiling run for collecting value profiles (or with assistance from programmer annotations [4, 7]). However, our study shows that opportunities exist for online identification of regions and it is also feasible to do such identification at runtime. Future work will be targetted at collecting value profiles at runtime with low overheads and using these to identify reuse regions at runtime.

5 Conclusions

This paper performs an empirical evaluation of computation reuse, with the focus on whether the reuse can be exploited profitably in a dynamic optimization setting. Our findings suggest: **(i)** programs contain many large regions, **(ii)** there is significant program reuse even at large reuse thresholds, which indicates that the regions could likely be exploited in an online setting, **(iii)** algorithms to identify reuse regions should be biased towards building contiguous regions and build non-contiguous regions only when the expected benefits are significant, and **(iv)** employing specialization along with memoization is likely to yield greater speedups than memoization alone because many regions have only one input vector, and because non-contiguous regions require runtime code generation.

References

[1] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57. ACM Press, 1996.

[2] J. L. Bentley. *Writing Efficient Programs*. Prentice Hall, Englewood Cliffs, 1982.

[3] D. Connors and W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 158–169, 1999.

[4] Charles Consel, Luke Hornof, Francois Noel, Jacques Noye, and Nicolae Volanschi. A Uniform Approach for Compile-time and Run-time Specialization. Technical Report RR-2775, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[5] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[6] A. Gonzalez, J. Tubella, and C. Molina. Trace-Level Reuse. In *Proceedings of the the International Conference on Parallel Processing*, September 1999.

[7] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, March 1997.

[8] Jian Huang and David J. Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 106–114, Orlando, Florida, January 9–13, 1999. IEEE Computer Society TCCA.

[9] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[10] Todd B. Knoblock and Erik Ruf. Data Specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, Pennsylvania, 21–24 May 1996.

[11] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[12] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–147, 1996.

[13] M. U. Mock and C. Chambers and S. J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *To appear in the Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, December 2000.

[14] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[15] Robert Muth, Scott Watterson, and Saumya Debray. Code Specialization Based on Value Profiles. In *Proceedings of the* 7^{th} *International Static Analysis Symposium (SAS 2000)*, pages 340–359. Springer LNCS vol. 1824, June 2000.

[16] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 109–121, New York, June 15–18 1997. ACM Press.

[17] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 248–258, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[18] Yiannakis Sazeides and James Smith. Implementation of Context-based Value Predictors. TechReport ECE-97-8, University of Wisconsin-Madison, December 1997.

[19] Timothy Sherwood and Brad Calder. Time Varying Behavior of Programs. TechReport CS99-630, University of California-San Diego, August 1999.

[20] James E Smith, Subramanya Sastry, Timothy Heil, and Todd Bezenek. Achieving High Performance via Co-Designed Virtual Machines. In *International Workshop on Innovative Architecture*, October 1999.

[21] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 194–205, June2–4 1997.