Programmer Specified Pointer Independence

Abstract

Good alias analysis is essential in order to achieve high performance on modern processors, yet precise interprocedural analysis does not scale well. We present a source code annotation, `#pragma independent`, which provides precise pointer aliasing information to the compiler, and describe a tool which highlights the most important and most likely correct locations at which a programmer should insert these annotations. Using this tool we perform a limit study on the effectiveness of pointer independence in improving program performance.

1. INTRODUCTION

Alias analysis, the identification of pointers which point to the same memory space, is an important part of any optimizing compiler. While numerous static alias analysis techniques exist (see [12] for a review), any static, intra-procedural analysis will be limited by its lack of knowledge of whole program behavior. However, it is possible for the programmer to provide this whole program knowledge by annotating the program suitably. An example of such an annotation is the `restrict` type qualifier that was introduced in the ANSI C99 standard [2]. In this paper, we investigate an alternative annotation that more closely maps to how pointer aliasing information is used by optimization passes within a compiler. Section 3 describes the semantics of and motivation for our new `#pragma independent` annotation. The implementation details for including the pragma in the compiler are described in Section 4.

In Section 5 we present a semi-automated system for assisting programmers in appropriately annotating source code. The system uses static information (derived from code structure) and dynamic information (derived from profiling) to suggest to the programmer independent pointer pairs with a high payoff for improving program performance.

To estimate the efficacy of the pragma and our tool, we present performance numbers in Section 6. We evaluate the impact of the pragmas using gcc for both a simulated in-order single issue processor and the EPIC Intel Itanium processor. We perform a second evaluation using the CASH experimental compiler [3], targeting a reconfigurable architecture. In addition, we evaluate the ability of the tool to identify important pragmas for performance improvement and the amount of effort involved in validating the automatically discovered pragmas.

In Section 7 we conclude that programmer specified pointer independence is a reasonable and effective scheme for improving program performance, but both the compiler and the targeted hardware must have the means of taking advantage of the improved aliasing information.

2. RELATED WORK

Pointer analysis is an important part in any optimizing or parallelizing compiler as potentially aliasing memory references can introduce false dependencies which inhibit optimizations and thread creation. While much work has been done to improve the precision and efficiency of pointer analysis [12], an intra-procedural static pointer analysis can not take advantage of whole program, dynamic information. Inter-procedural pointer analysis performs a whole program analysis, but does to scale with program size without losing precision [13, 26] and is complicated by separate compilation and the use of library functions. Using our methodology, the programmer provides pointer independence information which the compiler uses directly, just as it would use the results of a complex and expensive alias analysis. The overhead in the compiler of supporting our method is therefore virtually nonexistent.

Previous systems have used programmer annotations to provide memory aliasing information to the compiler or to analysis tools. In these systems the annotation is a type qualifier and the purpose is to aid in program understanding [1], program checking and verification [7, 8], or supporting type-safety [11]. In contrast, our annotation is not a type, but a precise statement of pointer independence. The compiler has no obligation to ensure the correctness of the annotations and the purpose of the annotations is simply to increase optimization opportunities and application performance. The ANSI C99 `restrict` type qualifier was designed to promote optimization [2], but does not directly map to information immediately usable by an optimizing compiler. The SGI MipsPro compiler provided an `ivdep` pragma which is used to break loop-carried dependencies between memory references in an inner loop. We describe a much more general approach. ASAP [15] is a language for describing the aliasing properties within data structures. ASAP also relies upon the programmer to ensure correctness.

Another solution to the problem of overly conservative alias information is performing dynamic disambiguation at run-time. This

```
void example(int *a, int *b, int *c)
{
#pragma independent a b
#pragma independent a c
    (*b)++;
    *a = *b;
    *a = *a + *c;
}
```

Figure 1: Pragma usage example. The programmer has informed the compiler that the pairs (a,b) and (a,c) will never alias, but makes no claims about the relationship between b and c.

can either be done completely in the compiler by generating instructions to check addresses [20] or by a combination of compiler and hardware support [21, 18]. Hardware support allows the compiler to speculatively execute instructions under the assumption that memory references do not alias. If the assumption proves false, potentially expensive fix-up code must be executed. A hardware based solution has the added advantage over both traditional pointer analyses and our approach in being able to successfully optimize cases where pointers do alias, but only infrequently. On the other hand, our proposal requires no special hardware and the final executable contains no extra instructions to check for aliasing.

Previous work has evaluated the effect of improved alias analysis on program performance [9, 14, 6, 10]. The aggressive register promotion and memory redundancy elimination of CASH parallel these studies.

The main contribution of this work is the semi-automated toolflow for independent pointer discovery. This tool-flow uses both program structure information and run-time execution statistics to propose "interesting" pointer pairs to be further investigated by the programmer. We show that the tool is very focused and requires little programmer intervention to obtain most of the benefits of the precise aliasing information.

3. #PRAGMA INDEPENDENT

We propose a pragma which allows the programmer to provide the compiler with precise and useful pointer independence information. The pragma has the syntax:

`#pragma independent` *ptr1 ptr2*

This pragma can be inserted anywhere in the program where *ptr1* and *ptr2* are both in scope. The pragma guarantees to the compiler that, within the scopes of *ptr1* and *ptr2*, any memory object accessed using *ptr1* will be distinct from any memory object that is accessed using *ptr2*.

We also allow the use of the pragma with n arguments, where $n > 2$; this implies pairwise independence between all pointer pairs from the argument list. Since the multiple-argument form does not provide increased expressive power (it merely reduces the number of annotations required), it will not be discussed further.

As an example, consider the C code in Figure 1. We assume that pairwise independence exists between the pairs (`a`,`b`) and (`a`,`c`) but nothing can be said about the relationship between `b` and `c`. The Itanium assembly code generated from this example by a pragma-aware `gcc` is shown in Figure 2. Using the additional pointer independence information, the compiler can successfully remove an unnecessary store to `a`.

The independence pragma is easy to use and reason about, since the programmer only has to take into account the behavior of two pointers. Furthermore, this type of information is exactly what an optimizing compiler needs when performing code motion optimizations such as partial redundancy elimination (PRE) and instruction scheduling.

4. USING #PRAGMA INDEPENDENT IN THE COMPILER

We have added support for the independence pragma to both `gcc` (version 3.3) and to the Compiler for Application Specific Hardware (CASH). Within `gcc`, we have modified the front-end to parse `#pragma independent` and, for each pointer variable declaration, maintain a list of pointer variables which have been declared as independent of that pointer. Within the alias analysis initialization phase of the `gcc` back-end, we then propagate this information to compiler temporaries. Since independent pointers must point to completely independent memory objects, we also propagate the independence information through address calculations. For example, `p` and `p+3` are assumed to point within the same "object", and thus the independence information valid for `p` is assumed to be valid for `p+3` as well. Also, if `p` is assigned to `q`, we propagate whatever independence information we have from `p` to `q` as well. Finally, when `gcc`'s optimization passes query for pairwise pointer independence, we use the independence information if possible. Pointer independence information is used by `gcc` in the CSE/PRE and instruction scheduling passes. Unfortunately, `gcc` does not have a register promotion optimization pass, which has been shown to benefit significantly from improved pointer independence information [19, 22]. Overall, relatively little code is needed to add full support for the independence pragma to a conventional compiler (less than 200 lines of code).

Within CASH the Suif [25] front-end uses `#pragma independent` statements to annotate to the corresponding variable symbols. We then run a dataflow analysis that propagates the independence information through compiler temporaries and pointer expressions. In CASH, may-dependencies between memory operations are first-class objects, represented by token edges [4]. The memory disambiguation pass removes token edges between memory references that it can prove do not alias; the disambiguator was modified to query independence pragma information. The compiler can then take advantage of the increased parallelism in the dependency graph.

5. AUTOMATED ANNOTATION

We have developed two systems for partially automating the annotation of source code with independence pragmas. Both systems combine a compiler's static analysis with runtime information to provide a ranked list of pairs of pointers that are candidates for being marked as independent. It is then the programmer's responsibility to evaluate these candidates for correctness.

Figure 3 shows a code snippet which has been automatically annotated with candidate independence pointer pairs. The scores heuristically estimate the effect that making the pair independent will have on improving program performance. These scores, as described below, summarize both information about the static code structure and execution frequencies. The pair (`arr1`,`arr2`) has

Without pragma

```
ld4 r14 = [r33]    // r14 = *b;
;;
adds r14 = 1, r14  // r14++;
;;
st4 [r33] = r14    // *b = r14;
st4 [r32] = r14    // *a = r14;
ld4 r15 = [r34]    // r15 = *c;
;;
add r14 = r14, r15 // r14 += r15;
;;
st4 [r32] = r14    // *a = r14;
br.ret.sptk.many b0
```

With pragma

```
ld4 r14 = [r33]    // r14 = *b;
;;
adds r14 = 1, r14  // r14++;
;;
st4 [r33] = r14    // *b = r14;

ld4 r15 = [r34]    // r15 = *c;
;;
add r14 = r14, r15 // r14 += r15;
;;
st4 [r32] = r14    // *a = r14;
br.ret.sptk.many b0
```

Figure 2: Itanium assembly code generated from the source in Figure 1. The double semicolons separate concurrent instruction groups. Using the information from the independence pragma, the compiler can remove a store instruction. On the Itanium processor, this avoids a split issue in the third instruction group, reducing the cycle time of the function.

```
void
array_sum(int *arr1, int *arr2, int n,
          int *result)
{
#pragma independent arr1 result /* score: 15 */
#pragma independent arr2 result /* score: 12 */
#pragma independent arr1 arr2 /* score: 1100 */
    int i, sum = 0;

    for(i = 0; i < n; i++)
    {
        *arr1 += *arr2;
        sum += *arr2;
    }
    *result = sum;
}
```

Figure 3: Sample code with pragma annotations and scores as produced by our tool-flow.

a much higher score than the other two pairs since these pointers are both accessed within the loop body. Knowing that they are independent allows the compiler to load the values of `*arr1` and `*arr2` into registers for the whole loop execution (perform register promotion). The pair (`arr1`,`result`) has a higher score than the pair (`arr2`,`result`), reflecting the fact that there is an opportunity to schedule the stores to `arr1` and `result` in parallel after register promotion.

The code fragment in Figure 3 was automatically annotated by using the tool-flow depicted in Figure 4. Of course, nothing prevents the function `array_sum` from being called with pointers that point to overlapping memory regions as the arguments `arg1` and `arg2`. Although the tool-flow checks whether this ever occurs for the profiling input sets, this is no guarantee of code correctness. It is the responsibility of the programmer to verify the correctness of the annotations by inspecting all the call sites of `array_sum`. The annotation scores serve as a heuristic to the programmer, focusing attention on the pairs which are most likely to bring performance benefits. As we show in Section 6, the scores closely track the 90-10 rule of program hot-spots (there are very few hot annotations) and verifying an annotation is not a time consuming task. Programmer effort for checking annotations is thus focused on the important pairs.

CASH Within CASH, memory dependencies are first-class objects represented by token edges [4] (see Figure 5). A memory disambiguation pass examines every token edge and, using traditional intra-procedural pointer analysis, eliminates edges between memory reference that can be proven not to alias. The base pointers of memory references that the analysis determines might alias are marked as candidates for annotation. Every pair of pointers is associated with a *static score* which estimates, for that pair, the benefit of declaring that pair as independent. For example, a pair which prevents an important optimization from taking place would receive a high score. There are many possible ways to compute a relevant score using only static information. Our current implementation uses a simple heuristic: the score of `pragma independent a b` is the number of token edges between memory accesses to `a` and `b`.

gcc `gcc` statically scores each pointer pair with the number of times `gcc`'s optimization passes query for independence information between the two pointers.

Although in many cases the two compilers find the same candidate pointer pairs, in some cases each compiler will find a pair the other does not. The CASH compiler is better at finding pointer pairs which are inhibiting optimizations, especially when the pointers are separated by complicated control flow. The `gcc` compiler will sometimes find pointers which CASH, with its more sophisticated alias analysis, can identify as not aliasing or not impacting optimization (if both pointers are used only in loads, for example).

Since pairs are aggressively generated without using inter-procedural analysis, some pairs will alias at run-time and therefore should not be annotated as independent. Thus, both CASH and `gcc` also instrument the program to collect run-time information: for each pointer pair, a special check operation is inserted. This check operation acts as both an *aliasing check* and a *frequency counter*. When the program is run, the check records any pointer pairs that alias, and thus are not independent. The frequency counter is used to identify frequently-executed code. With CASH, the check operation is an instruction implemented directly within the simulator whereas with `gcc` it is a function call to a library function performing a dynamic check.

The compile-time and run-time information are combined by a script, which weeds out the pairs which were discovered to alias and computes an overall score using both the static score and the frequency counts for each pair (currently by multiplying them). The script sorts the annotations by the overall score, and can optionally annotate the original source code with the annotations whose scores

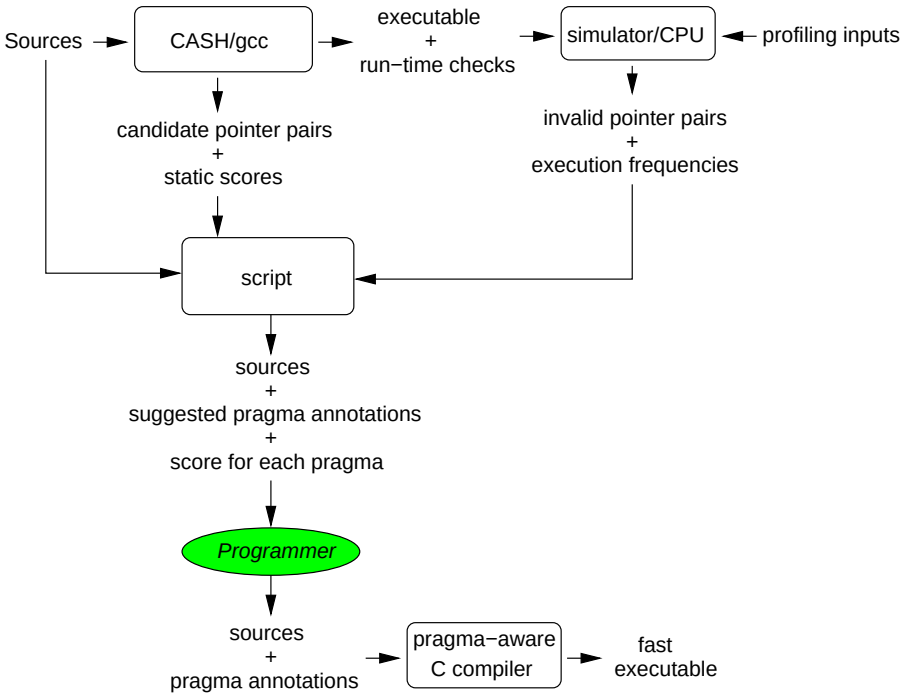

Figure 4: Tool-flow for independence pragma source annotation. Notice that the programmer is part of the tool-flow, certifying the correctness of the suggested annotations.

are above a certain programmer-selected threshold; this is how the code in Figure 3 was produced. The programmer's effort can then be focused on analyzing the source code having pairs with high overall scores.

6. RESULTS

6.1 Evaluation

We have evaluated the effectiveness of our automated annotation system and the ability of the modified compilers to take advantage of the independence information on three very different machine models: (1) We used our modified version of `gcc` to compile to the MIPS-like SimpleScalar [5] architecture which we then simulated running on an in-order, single issue processor. (2) We used the same `gcc` version to compile for a 733Mhz EPIC Intel Itanium processor [16]. Programs were compiled using the optimization flags `-O2 -funroll-loops`. (3) Finally, we used our CASH compiler to target a reconfigurable fabric. Our results are obtained from the programs in Mediabench [17], Spec95 [23], and Spec2000 [24]. When possible we ran the annotation tool on the training sets and collected performance results from the reference sets. Both the CASH and `gcc` generated annotations were used for the reconfigurable target while only the `gcc` generated ones were used for the SimpleScalar and Itanium targets.

Our two simulators provide cycle-accurate measurements, but are about three orders of magnitude slower than native execution. The measurements on the real Itanium system are plagued by variability from low-resolution timers and system activity. We have thus used different input sets for the simulated and real system (short ones on simulators, large ones on the real system). In addition, we do not produce results for large benchmarks with the simulators nor results from smaller benchmarks on the Itanium.

The source code of all benchmarks has been annotated with independence pragmas using the automated system. Although we have inspected and verified some benchmarks, we have not manually inspected every individual pragma that the system produces. All benchmarks produce the correct output when run with the annotations.

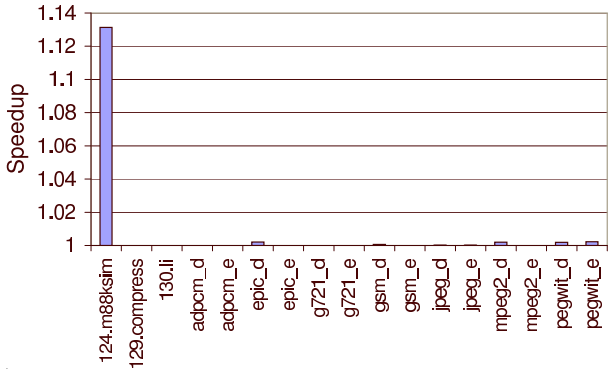

Figure 6: Speed-up due to pragmas on an in-order processor. Code is compiled with `gcc` for a simulated in-order, single issue processor.

6.2 Speed-ups

The execution speed-up for annotated code on the in-order, single issue simulated processor is shown in Figure 6. As expected, the effect of the independence pragmas is mostly negligible. This architecture is incapable of taking advantage of additional memory parallelism. Furthermore, the `gcc` SimpleScalar PISA backend is somewhat rudimentary. Few target specific optimizations

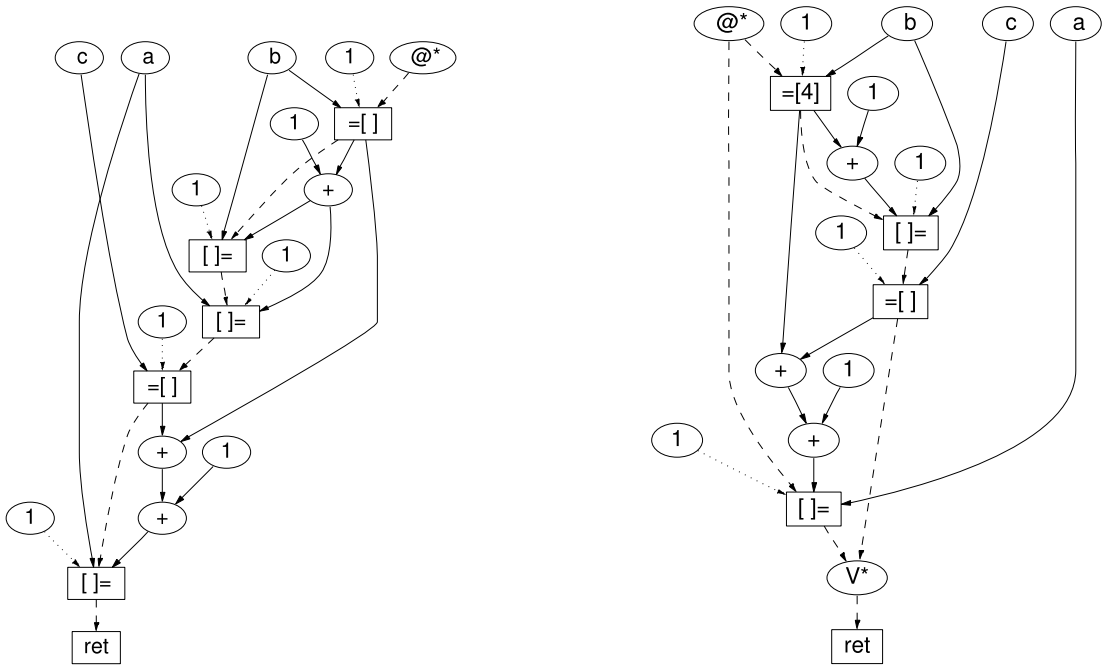

Figure 5: CASH internal representation of Figure 1. On the left is the code with no pragmas while on the right is the code with pragmass. Load operations are shown as `=[]`; each has 3 inputs: address, predicate, and token and two outputs: data and token. Store operations are shown as `[]=`; each has an 4 inputs: address, data, token, and predicate, and one token output. Dashed lines represent tokens (memory dependences); "@*" is the "input token". The V node "joins" tokens. Dotted lines represent predicates. Note that the graph on the right has one fewer store.

are performed and the underlying machine model does not accurately or precisely describe the actual machine model. Even so, `124.m88ksim` demonstrates a 1.13 speed-up using the pragmas. Most of the remaining benchmarks either show little or no improvement. A couple of benchmarks, `mpeg2_e` and `gsm_e` exhibit a small slowdown. The reason for the slowdown is that `gcc`'s scheduler uses a simplistic and inaccurate machine model.

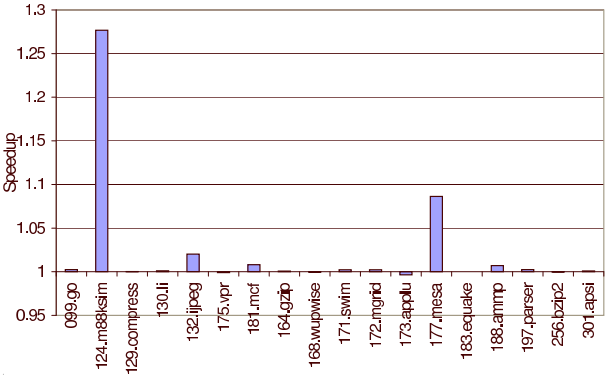

Figure 7: Speed-up using `#pragma independent` annotated code compiled with `gcc` for an Intel Itanium processor.

The execution speed-up for annotated code on the Itanium is shown in Figure 7. As expected, the highly parallel Itanium processor does better than the in-order SimpleScalar processor. `124.m88ksim` shows a speed-up of 1.28, `177.mesa` a speed-up of 1.09, and `132.ijpeg` a speed-up of 1.02. The remaining benchmarks either did not show a significant speed-up, or had too short a running time to be measured precisely. The observed improvements appear to be exclusively a result of improved instruction scheduling suggesting a more sophisticated Itanium compiler would demonstrate more benefit (as in [10]). Furthermore, there are only a few critical pragmas in each benchmark. For example, in `124.m88ksim` there is just one pragma that accounts for all the observed speed-up; it breaks dependencies within a memcpy-like loop. In `177.mesa`, three pragmas inside a critical function with pointer arguments are enough to account for all of the speed-up.

The execution speed-up for annotated code compiled for a reconfigurable fabric is shown in Figure 8. Most benchmarks demonstrate meaningful speed-ups with the most significant being speed-ups of 2.00, 1.93, 1.74, 1.35 for `adpcm_d`, `jpeg_d`, `adpcm_e`, and `gsm_d` respectively.

6.3 Scoring

One goal of our tool is to give the programmer a way to pass information to the compiler without increasing the programming burden. To this end we evaluated the effectiveness of our tools in guiding the programmer effort toward the most profitable code regions.

Histograms of the scores of the pragmas, like the one in Figure 9 for `132.ijpeg`, look surprisingly similar. Figure 9 shows histograms of both the static and dynamic scores. The *x* axis is the normalized score of an annotation, binned in 20 equal intervals. The *y* axis represents the number of annotations which have a score within 5% of the *x* value. For example, the 5% bar labeled "dynamic", with a value of 3, shows that 3 annotations have a score between 5% and 10% of the maximum score found. Both distributions have a sharp

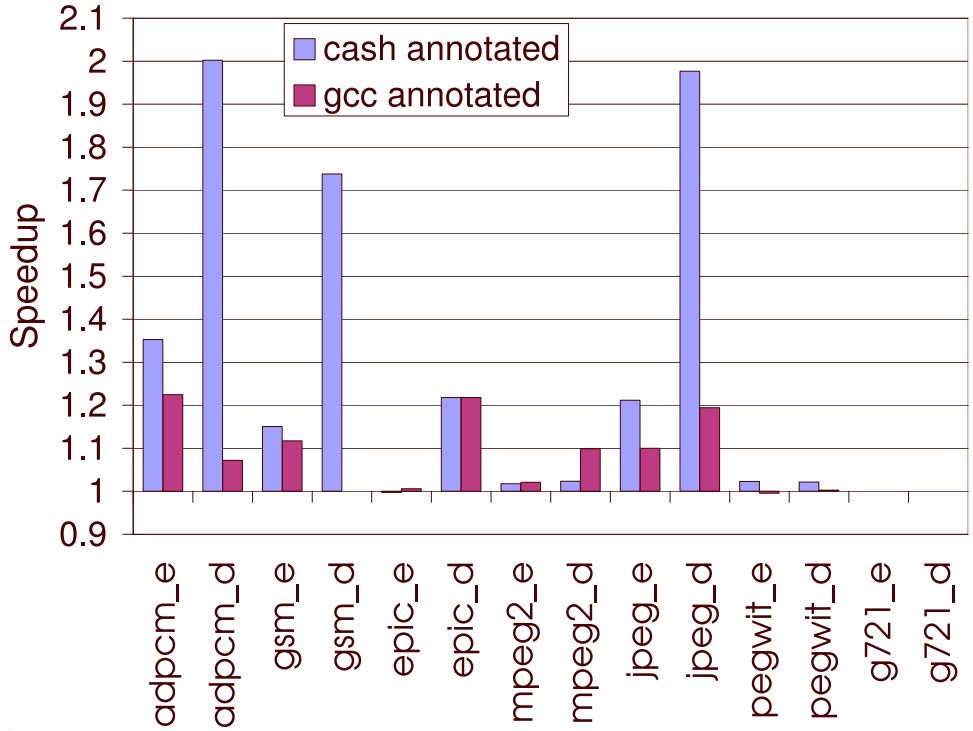

Figure 8: Speed-up using `#pragma independent` annotated code compiled with the CASH compiler for a simulated reconfigurable fabric.

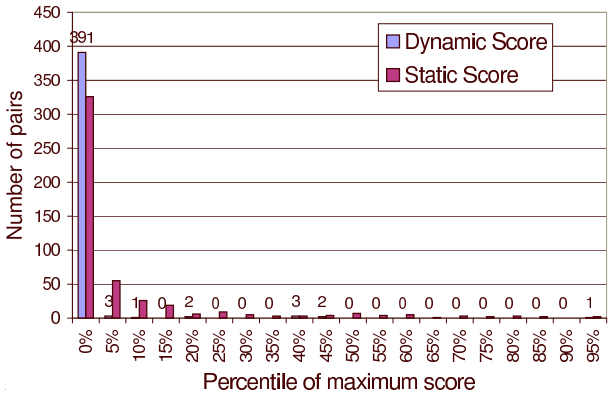

Figure 9: Score histogram for `132.ijpeg`.

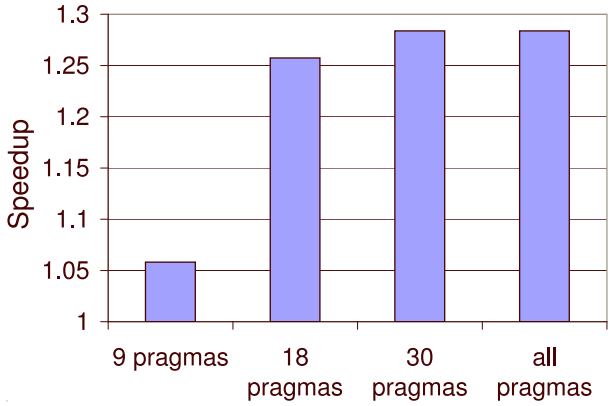

Figure 10: Speed-up for `132.ijpeg` run on a simulated reconfigurable fabric as a function of the number of pragmas used. Pragmas were added in order of decreasing importance.

knee, which suggests a cut-off point for useful annotations. In this example, 22 pragmas account for the top 96% of the scores. These are the most likely to require the attention of the programmer. And in fact, as Figure 10 shows, the top 18 pragmas account for almost all of the improvement that can be gained.

In Table 1 we give the pragma counts found by our `gcc`-based automatic instrumentation system. The first three columns show the total number of pragmas inserted, the number of pointer pairs which were executed at least once for the given input set, and the number of pairs which were found to alias, thus whose annotations are incorrect. The fourth column shows how many of the correct annotations are below the "knee" of the curve (these were manually estimated by looking at the score distribution).

In order to verify that the high scoring annotations are indeed the most important we have carried out two experiments. We annotate each program with only a small number of annotations, the ones with the highest scores. Figure 11 presents results for all benchmarks for which we manually inspected only the highest ranking annotations (see below).

6.4 Validation

We tested our claim that manually validating an automatically generated pragma is not an onerous task (even if the programmer is unfamiliar with the code) by having several programmers verify

Bench	total	checked	conflict	useful
124.m88ksim	119	57	2	12
129.compress	3	3	0	6
130.li	56	21	3	6
132.ijpeg	490	142	8	22
134.perl	744	267	42	22
175.vpr	188	39	4	12
181.mcf	132	60	7	14
adpcm_d	12	3	0	6
adpcm_e	12	3	0	6
epic_d	41	11	7	11
epic_e	32	22	3	13
g721_d	0	0	0	0
g721_e	0	0	0	0
gsm_d	36	10	1	9
gsm_e	36	21	4	11
jpeg_d	418	90	2	12
jpeg_e	453	68	9	10

Bench	total	checked	conflict	useful
mesa	979	107	9	25
mpeg2_d	94	64	0	3
mpeg2_e	72	21	4	9
pegwit_d	34	24	3	11
pegwit_e	34	25	4	14
176.gcc	3470	2406	504	44
197.parser	159	144	38	12
256.bzip2	40	36	34	3
300.twolf	451	173	52	27
168.wupwise	3	3	0	3
171.swim	0	0	0	0
172.mgrid	7	7	1	5
173.applu	2	2	2	0
177.mesa	950	94	8	37
183.equake	30	13	2	6
188.ammp	252	82	11	11
301.apsi	463	362	3	14

Table 1: Annotations statistics. The columns represent: benchmark name, total pointer pairs instrumented, pointer pairs with at least one run-time check, pointer pairs found to alias at run-time, number of most likely useful pointer pairs (knee of histogram curve). The last column is the number of pragmas that would most likely bring the highest benefit, and would thus be the best targets for manual validation.

	local	argument	global
local	30	51	24
argument		92	29
global			9

Table 3: The types of pointer pairs verified. Of the 235 pairs that were manually inspected, we classified each verified pair by the defining scope of its members. A local pointer variable was always counted as a local reference, even if it could be proved to always equal a pointer argument.

some of the annotated source. The programmers were instructed to verify only the highest scoring annotations.

Even though they had little experience with the code, it took an average of less than 2 minutes per pragma to validate its correctness. Most of the time was spent exploring the call tree to determine the origins of pointer arguments to functions. In addition, as the programmer became more acquainted with the structure and conventions of a benchmark, validation took less time. Some program constructs made validation difficult or impossible. In this case the annotation was marked as incorrect. We expect that a programmer with a deeper understanding of the code would be able to verify annotations almost instantaneously.

Table 3 shows the counts for the different relationships between the verified pointers. Not surprisingly, few pointer pairs were between global references as the compiler can almost always differentiate the pointers in this case. Without inter-procedural analysis, function arguments, and local variables whose values come from function arguments, are the most likely candidates for programmer specified pointer analysis.

7. CONCLUSION

Uncertainty about pointer relationships and the inability to perform whole program analysis frequently handicaps compiler optimizations, particularly for languages like C. However, it is frequently the case that the programmer has knowledge about pointers which could help the optimizer, but the language provides no convenient mechanism for expressing this type of information. In this paper we have presented a mechanism which enables the programmer to specify to the compiler that certain pointers access disjoint memory regions and quantified the benefits that can be derived from exploiting this mechanism. We have also presented a tool-chain which uses the compiler optimizer and run-time information to suggest to the programmer a small number of pointer pairs whose known non-aliasing could have a big impact on the program performance. While these suggested pairs require manual verification, we have shown that even an investment of 10 minutes of code analysis can provide significant benefits. Allowing programmers to provide pointer independence information can result in meaningful increases in performance provided that the compiler and targeted architecture a capable of taking advantage of such information. We conclude that programmer specified pointer independence is a scalable, effective alternative to inter-procedural pointer analysis.

8. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), November 2002.*, 2002.

[2] ANSI. Programming languages - C, 1999.

[3] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, Montpellier (La Grande-Motte), France, September 2002.

[4] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation*

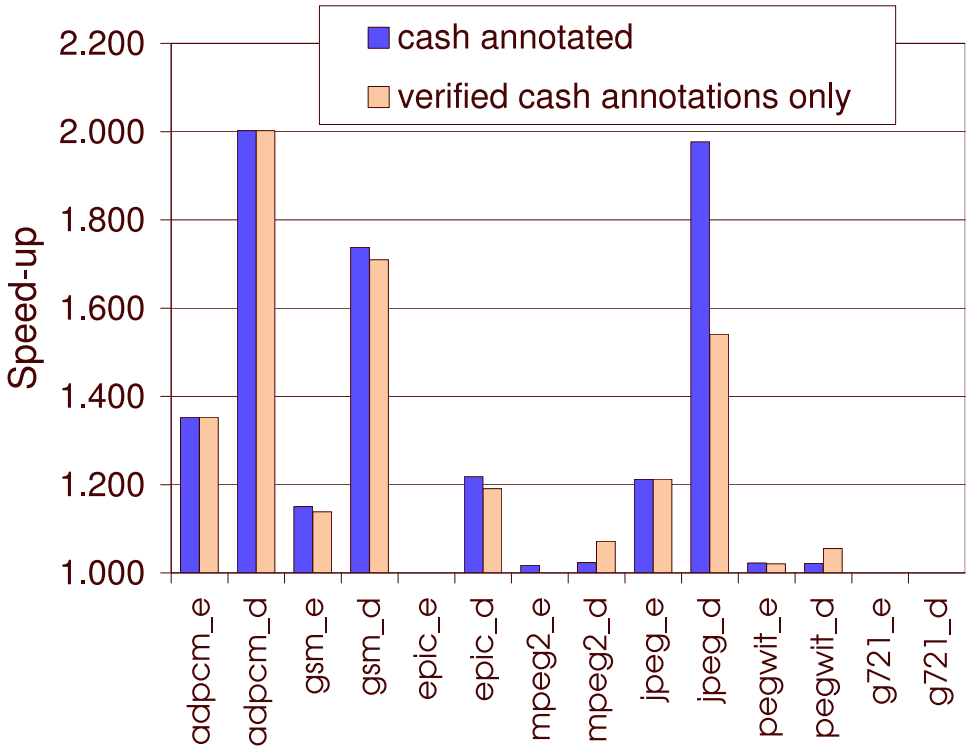

Figure 11: Speed-up due to pragmas on the reconfigurable hardware platform. First bar uses all pragmas (and is potentially unsafe); the second bar uses the highest-ranking pragmas which were manually certified to be safe.

and Optimization (CGO 03), San Francisco, CA, March 23-26 2003.

[5] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25 (3), pages 13–25. ACM SIGARCH, June 1997.

[6] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69. ACM Press, 2000.

[7] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[8] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[9] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 121–133. ACM Press, 1998.

[10] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 47–58. ACM Press, 2001.

[11] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[12] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[13] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[14] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.*, 39(1):31–55, 2001.

[15] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 208–216, Cancun, Mexico, April 1994. IEEE Computer Society.

[16] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, 2002.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In

Bench	lines	total	inspected	correct	time (min)	time/pragma
adpcm_d	307	8	8	8	5	<1
adpcm_e	307	8	8	8	2	<1
gsm_e	5841	33	9	9	25	2.8
gsm_d	5841	36	5	5	9	1.8
epic_d	2528	9	9	9	33	3.7
epic_e	2830	21	21	20	11	<1
mpeg_d	10596	73	58	34	113	1.9
mpeg_e	7791	63	10	10	8	<1
jpeg_e	27496	264	32	31	90	2.8
jpeg_d	27496	248	13	13	10	<1
pegwit_e	6944	99	26	26	24	<1
pegwit_d	6944	98	26	26	2	<1
mesa	67081	361	30	27	16	<1

Table 2: Manual verification effort, expressed in minutes. For each benchmark we list the number of lines of source code, the number of automatically generated pragmas, the number of pragmas manually inspected (if there were many generated pragmas, only the highest ranking were inspected), the number of pragmas ascertained correct, and the times spent on validation: overall, and per-pragma.

Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture, pages 330–335, 1997.

[18] Jin Lin, Tong Chen, Wei-Chung Hsu, and Pen-Chung Yew. Speculative register promotion using advanced load address table (alat). In *International Symposium on Code Generation and Optimization*, pages 125–133, 2003.

[19] John Lu and Keith D. Cooper. Register promotion in C programs. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319. ACM Press, 1997.

[20] Alexandru Nicolau. Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):633–678, 1989.

[21] Matt Postiff, David Greene, and Trevor N. Mudge. The store-load address table and speculative register promotion. In *International Symposium on Microarchitecture*, pages 235–244, 2000.

[22] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *Proceedings of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation*, pages 15–25. ACM Press, 1998.

[23] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[24] Standard Performance Evaluation Corp. *SPEC CPU2000 Benchmark Suite*, 2000.

[25] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[26] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–, 1995.