Programmer Specified Pointer Independence

David Koes Mihai Budiu Girish Venkataramani
Seth Copen Goldstein

April 14, 2003
CMU-CS-03-128

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Good alias analysis is essential in order to achieve high performance on modern processors, yet interprocedural analysis does not scale well. We present a source code annotation, `#pragma independent`, which is a more flexible, intuitive and useful way for the programmer to provide pointer aliasing information than the current C99 `restrict` keyword. We describe a tool which highlights the most important and most likely correct locations at which a programmer can insert the pragmas. We show that such annotations can be used effectively in compilers to achieve speedups of up to 1.2x.

This research was funded by National Science Foundation grants no. CCR-9876248 and no. IIS-0117658 and a Hewlett-Packard project funded by the Defense Advanced Projects Agency. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect those of the sponsors.

Keywords: compilers, pointer analysis, user annotations

1 Introduction

Alias analysis, the identification of pointers which point to the same memory space, is an important part of any optimizing compiler. While static alias analysis techniques exist (see [10] 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 propose an alternative annotation that is simultaneously more powerful, flexible, and intuitive than the `restrict` keyword. Section 3 describes the semantics of and motivation for our new `#pragma independent` annotation. Section 4 compares our pragma to the ANSI C99 `restrict` keyword.

In Section 5 we present a semi-automated system for assisting programmers in appropriately annotating their source code. In this system, the compiler highlights pointer pairs whose aliasing relationship cannot be statically determined by an intra-procedural analysis, but whose non-aliasing would enable other optimizations, and instruments the executable with run-time checks for aliasing. The executable is then run on a sample input. The pointer pairs which did not exhibit run-time aliasing are then ranked using both static, compile-time information and dynamic, run-time information. This ranking focuses the attention of the programmer on those pointer pairs which, with high likelihood, can be correctly labeled independent with the greatest impact upon performance. The implementation details are described in Section 6.

In order to show the efficacy of the pragma and our tool, we present performance numbers in Section 7. We compile using a conventional compiler, gcc, targeting both a simulated in-order single issue processor and the EPIC Intel Itanium processor. We also use an experimental compiler, CASH [3], to target a reconfigurable architecture.

The true power of the pragma is hard to judge because of a chicken-and-egg problem: in order to evaluate the effectiveness of alias information, optimizations which take full advantage of such information are needed. However, such optimizations are only implemented in a compiler that already provides substantial alias information. For example, gcc does not have very sophisticated alias analysis and consequently does not fully implement optimizations such as register promotion which would benefit greatly from improved alias information. Therefore, it is not too surprising that for many benchmarks gcc cannot produce a significant performance improvement using the information provided by the annotations while a more modern compiler can. Even given these limitations, the use of independence pragmas can result in more than 20% improvement for some benchmarks.

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 [10], 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 fails to scale well as program size increases [11, 22] and is complicated by separate compilation and the use of library functions. In our method, 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 [6, 8], or supporting type-safety [9]. In

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

Figure 1: An example where `restrict` can not be used, but code generation benefits from the use of the independence pragma.

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 has shortcomings which are addressed more fully in Section 4.

Another solution to the problem of overly conservative alias information is doing dynamic disambiguation at run-time. This can either be done completely in the compiler by generating instructions to check addresses [16] or by a combination of compiler and hardware support [17, 14]. 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.

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 intersection of the scopes of *ptr1* and *ptr2*, any memory object accessed using *ptr1* will be distinct from any memory object that is accessed using *ptr2* and vice versa.

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 (except reducing the number of annotations required), it will not be discussed further.

As an example, consider the C code in Figure 1; pairwise independence exists between the pairs (`a`,`b`) and (`a`,`c`) but nothing can be said about the relationship between `b` and `c`. We have modified a recent version of gcc targeting the Itanium architecture to understand and take advantage of the `independence` pragma. The assembly code generated from this example 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. Contrast this to the `restrict` keyword, which implies a relationship

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: The generated Itanium assembly code for the source in Figure 1. 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.

between one pointer and all other pointers within the same scope (see the next section). 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 Comparison to `restrict`

The formal definition of `restrict` takes up a full page of the C99 specification, not including another page of usage examples. A simplified, but more rigorous and lengthy definition is given in [7]. Within gcc the definition is interpreted to mean that no two restricted pointers can alias, but a restricted pointer and an unrestricted pointer may alias. To correctly annotate a pointer declaration *p* with the `restrict` type qualifier, it is necessary for the programmer to ensure that *p* does not alias with *all* other restricted pointer declarations that are visible in the current scope. Unless `restrict` is only used sparsely, it becomes a significant burden to the programmer to correctly reason about its correct application. Using `#pragma independent` correctly, with its weaker but more precise semantics, requires the programmer to only reason about a single pair of pointers. In addition, the pragma is capable of representing pointer relationships that are not representable by `restrict`. For example, the pairwise independence of two pairs of pointers as in Figure 1.

Besides convenience to the programmer, the independence pragma is also easier to use by the compiler. The information provided by `restrict` does not directly map to the way conventional compilers use pointer alias information. Within an optimizing compiler, pointer analysis is mostly useful to determine that two pointers do *not* alias each other. While `restrict` can provide such pairwise information (if both pointers are restricted) it can only be used if both pointers also exhibit the much more restrictive property of not aliasing all restricted pointers. The independence pragma, by contrast, exactly maps to the internal application of pointer independence information within the compiler. Indeed, the semantics of the pragma were originally motivated by the needs of some optimizations of our CASH compiler.

Overall, the independence pragma is a more flexible, more intuitive, and more useful means of annotating source code to communicate pointer aliasing information to the compiler than `restrict`.

```
void summer(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.

5 Automated Annotation

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 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 above code fragment was automatically annotated by using the tool-flow depicted in Figure 4. Of course, nothing prevents `summer` 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, of course, no guarantee of the code correctness. It is the responsibility of the programmer to verify the correctness of the annotations by inspecting all the call sites of `summer`. The annotation scores serve as a heuristic to the programmer, focusing the attention on the pairs which are most likely to bring performance benefits. As we show in Section 7, the scores closely track the 90-10 rule of program hot-spots: there are very few hot annotations. Programmer effort is thus minimized.

The code instrumentation is performed with a modified version of gcc. gcc associates with each pair of pointers a *static score* which estimates the effect on optimizations of declaring that pointer pair as independent. Within gcc, pointer independence information is most useful for CSE/PRE and instruction scheduling operations. Unfortunately, gcc does not have a very robust register promotion optimization pass, which has been shown to benefit significantly from improved pointer independence information [15, 18]. Although there are many possible ways to compute a relevant score using only static information for a given pointer pair, our current implementation uses the simple, but effective, heuristic of counting the number of times gcc's optimization passes query for independence information between the two pointers.

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, gcc also instruments the program executable to collect run-time information: for each pointer pair, before every use of a pointer of the pair that is reachable by definitions of both pointers in the pair, gcc inserts both an *aliasing check* and a *frequency*

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

counter. When the executable is run, the check records any pointer pairs which alias, and thus are not independent. The frequency counter is used to determine pointers dereferenced in frequently-executed code.

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 frequencies 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 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. We show in Section 7 that the number of relevant annotations tends to be small even for large programs.

6 Implementation

We have added support for the independence pragma to both gcc and 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. Overall, relatively little code is needed to add full support for the independence pragma to a conventional compiler.

Within CASH, the processing of the pragma in the front-end follows the same flow as within gcc: the SUIF [21] front-end parses `#pragma independent` and applies it as an annotation to the corresponding variable declarations. 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]. A pointer 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 aggressively take advantage of the increased parallelism in the dependency graph since compiling to a reconfigurable fabric allows us to fully exploit parallelism.

7 Results

7.1 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 [12]. Programs were compiled using the optimization flags `-O2 -funroll-loops`. (3) Finally, we used our CASH compiler to target a simulated reconfigurable fabric connected to a realistic, bandwidth-limited memory system. Our results are obtained from the programs in Mediabench [13], Spec95 [19], and Spec2000 [20]. When possible we ran the annotation tool on the training sets and collected performance results from the reference sets.

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).[1]

The source code of all benchmarks has been annotated with independence pragmas using the our automated system. When possible, the alias checking phase of the annotation is performed using an input set that is different from the input set used to evaluate performance. We do not manually inspect each individual pragma that the system produces. However, all benchmarks produce the correct output when run with the annotations.

7.2 Speed-ups

The execution speed-up for annotated code on the in-order, single issue simulated processor is shown in Figure 5. The effect of the independence pragmas is mostly negligible. This is not surprising as this architecture is incapable of taking advantage of additional memory parallelism. Furthermore, the gcc SimpleScalar PISA back-end is somewhat rudimentary. Few target specific optimizations 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 small slowdowns. The reason for the slowdowns is the fact that gcc's scheduler uses a simplistic and inaccurate machine model.

The execution speed-up for annotated code on the Itanium is shown in Figure 6. As expected, the highly parallel Itanium processor does better than the in-order SimpleScalar processor. `124.m88ksim` show a

---

[1] Since the Mediabench benchmarks do not have large input sets, the real system measurements were too noisy to be included in this paper; we are currently upgrading the kernel on our Itanium machine to obtain access to high-resolution hardware timers, which will enable us to collect data for all benchmarks on all systems.

Figure 5: *Speed-up using* `#pragma independent` *annotated code compiled with gcc for a simulated in-order, single issue processor.*

Figure 6: *Speed-up using* `#pragma independent` *annotated code compiled with gcc for an Intel Itanium processor.*

speed-up of 1.27, `177.mesa` a speed-up of 1.08, 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 execution speed-up for annotated code compiled for a reconfigurable fabric is shown in Figure 7. Most benchmarks demonstrate meaningful speed-ups with the most significant being speed-ups of 1.28, 1.25, and 1.19 for `132.ijpeg`, `adpcm_e`, and `epic_d` respectively. These speed-ups are conservative:

Figure 7: *Speed-up using* `#pragma independent` *annotated code compiled with the CASH compiler for a simulated reconfigurable fabric. Speed-ups using just the few highest ranking independence pragmas are also shown.*

Figure 8: *Score histogram for* `132.ijpeg`.

a 2x speed-up can be achieved for `adpcm_d` (shown as `adpcm_d_hand`) when independence pragmas are added by hand. The pragmas necessary to achieve this speed-up were not found by our automated system, which is based on gcc, since the independence information would have not helped gcc optimize the code. To verify, this, we compiled the hand annotated source using gcc and observed no performance improvement.

Figure 9: *Speed-Up for* `132.ijpeg` *run on a simulated reconfigurable fabric as more high ranking pragmas are added.*

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
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: The columns represent: benchmark name, total pointer pairs instrumented, pointer pairs with non-zero run-time checks, pointer pairs found to alias at run-time, number of most likely useful pointer pairs (knee of histogram curve).

We expect that if the annotation system used the CASH compiler to find candidate pointer pairs, we would see a significant performance improvement over our current results.

7.3 Scoring

One goal of our tool is to give the programmer a way to pass information to the compiler without increasing the programming burden. We are thus evaluating the effectiveness of our tools in guiding the programmer effort toward the most profitable code regions.

We have plotted the pragma score histograms for all programs. The plots look surprisingly similar to each other; we are showing a representative one for `132.ijpeg` in Figure 8. We are showing both static and overall 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 knee, which suggests a cut-off point for useful annotations. For this example, only 12 dynamic (the sum of the labels between 5% and 100%) annotations have scores in the interval 5%-100% from the maximum score. These are the most likely to require the attention of the programmer.

In Table 1 we give the pragma counts found by our 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 9 shows how performance of `132.ijpeg` improves as we add more pairs, in order of decreasing score. Although 490 pairs were flagged as candidates by the tool, fewer than 30 of the highest ranking pointer pairs are necessary to achieve nearly the same speed-up as using all the pairs.

Figure 7 presents results for all benchmarks, comparing the impact of using all annotations with the impact of using only up to 10 (we use fewer annotations for the smaller programs). `132.ijpeg` is the only program which requires more than 10 annotations to attain the full benefit; in all other cases nearly the full benefit of the annotations can be realized using just a few of the highest ranking pointer pairs.

8 Conclusion

Uncertainty about pointer relationships and the inability to perform whole program analysis frequently handicap the 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 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. Allowing programmers to provide pointer independence information can result in meaningful increases in performance. Of course, the programmer must verify that such annotations are safe. We conclude that programmer specified pointer independence is a scalable, effective alternative to inter-procedural pointer analysis. Our modified version of gcc and the scripts to annotate source code can be found at `http://www.cs.cmu.edu/~phoenix/independence`.

9 Acknowledgments

Dan Vogel wrote the scripts which back-annotate the source files with pragma starting from the collected instrumentation.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding, 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 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] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[7] Jeffrey S. Foster and Alex Aiken. Checking programmer-specified non-aliasing.

[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] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone.

[10] 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.

[11] 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.

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

[13] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.