Design-Driven Compilation

Radu Rugina and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract. This paper introduces *design-driven compilation*, an approach in which the compiler uses design information to drive its analysis and verify that the program conforms to its design. Although this approach requires the programmer to specify additional design information, it offers a range of benefits, including guaranteed fidelity to the designer's expectations of the code, early and automatic detection of design nonconformance bugs, and support for local analysis, separate compilation, and libraries. It can also simplify the compiler and improve its efficiency. The key to the success of our approach is to combine high-level design specifications with powerful static analysis algorithms that handle the low-level details of verifying the design information.

1 Introduction

Compilers have traditionally operated on the source code alone, utilizing no other sources of information about the computation or the way the designer intends it to behave. But the source code is far from a perfect source of information for the compiler. We focus here on two drawbacks. First, the code is designed for efficient execution, not presentation of useful information. The information is therefore often *obscured* in the program: even though the code implicitly contains the information that the compiler needs to effectively compile the program, the information may be difficult or (for all practical purposes) impossible to extract. Second, the source code may be *missing*, either because it is shipped in unanalyzable form or because it has yet to be implemented.

The thesis of this paper is that augmenting the source code with additional design information can ameliorate or even eliminate these problems. The result is a significant increase in the *reach* of the compiler (its ability to extract information about the program and use this information to transform the program) and the *fidelity* with which the compilation matches the designer's intent. We call this new compilation paradigm *design-driven compilation*, because the design information drives the compiler's search for information.

We believe the key to the success of this approach is an effective division of labor between the designer and the compiler. The design information should take the form of intuitive, high-level properties that any designer would need to know to design the computation. This information must be augmented by

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 150–164, 2001.
© Springer-Verlag Berlin Heidelberg 2001

powerful analysis algorithms that automate the low-level details of verifying the design information and applying the design.

1.1 Design Conformance

The design information specifies properties that the program is intended to fulfill. This information is implicitly encoded in the program, and the compiler would otherwise need sophisticated program analysis techniques to extract it. But the explicit (and redundant) specification of design information provides a range of advantages. First, it clearly states the programmer's intent, and non-conformance to the design can be caught early by the compiler. Second, using design information at the procedure level provides modularity and enables local analysis, separate compilation, and library support. Third, it is easier for the compiler to check the design information than to extract it, making the compiler structure simpler and making the compiler more efficient.

Design information for procedures can be expressed using *procedure interfaces*. The interface for a procedure specifies the effects of that procedure with respect to a given abstraction. In this paper we present two kinds of procedure interfaces. Pointer interfaces specify how procedures change the points-to information. Region interfaces specify the regions of memory that the entire computation of each procedure accesses.

It is not practical to expect the programmer provide detailed information within procedures. But the correctness of the design information depends crucially on the low-level actions of the code within each procedure. Our approach therefore *combines design specifications with program analysis*. Design information specifies high-level properties of the program. Static analysis automatically extracts the low-level properties required to verify that the program conforms to its design. Design conformance is an attractive alternative to static analysis alone because of its range of benefits:

1. **Fidelity:**
   - *Faithful Compilation:* The design information enables the compiler to generate parallel programs that faithfully reflect the designer's high-level expectations.
   - *Enhanced Code Reliability:* Verifying that the code conforms to its design eliminates many potential programming errors. Subtle off-by-one errors in array index and pointer arithmetic calculations, for example, often cause the program to violate its region interface design.
   - *Enhanced Design Utility:* Our approach guarantees that the program conforms to its design. Designers, programmers, and maintainers can therefore rely on the design to correctly reflect the behavior of the program, enhancing the utility of the design as a source of information during the development and maintainance phases.
2. **Modularity:**
   - *Local Analysis:* The design information enables the compiler to use a *local* instead of a *global* analysis — each procedure is analyzed independently of all other procedures.

- *Separate Compilation:* The design information enables the compiler to fully support the separate analysis and compilation of procedures in different files or modules, and to fully support the use of libraries that do not export procedures in analyzable form.
- *Improved Development Methodology:* The design information allows the compiler to analyze incomplete programs as they are under development. The programmer can therefore start with the design information, then incrementally implement each procedure. At each step during the development, the analysis uses the design information to check that the current code base correctly conforms to its design. The overall result is early detection of any flaws in the design and an orderly development of a system that conforms to its design.
- *Enhanced Interface Information:* In effect, our approach extends the type system of the language to include additional information in the type signature of each procedure. The additional information formalizes an additional aspect of the procedure's interface, making it easier for engineers to understand and use the procedures.

3. **Simplicity:**
   The availability of design information as procedure interfaces significantly simplifies the structure of the compiler. It eliminates the interprocedural analysis, and replaces it with the much simpler task of verifying the procedure interfaces. Improvements include increased confidence in the correctness of the compiler, a reduction in the implementation time and complexity, and increased compiler efficiency.

1.2 Automatic Parallelization of Divide and Conquer Programs

We have applied design-driven compilation to a challenging problem: the automatic parallelization of divide and conquer programs [4]. The inherent parallelism and good cache locality of divide and conquer algorithms make them a good match for modern parallel machines, with excellent performance on a range of problems [2,10,6,3].

The tasks in divide and conquer programs often access disjoint regions of the same array. To parallelize such a program, the compiler must precisely characterize the regions of memory that the complete computation of each procedure accesses. But it can be quite difficult to extract this information automatically. Divide and conquer programs use recursion as their primary control structure, invalidating standard approaches that analyze loops. They also tend to use dynamic memory allocation to match the sizes of the data structures to the problem size. The data structures are then accessed via pointers and pointer arithmetic, which complicates the analysis of the memory regions accessed by the computation of each procedure.

In our approach, the designer provides two kinds of design information: pointer information that summarizes how each procedure affects the points-to relationships, and region information that summarizes regions of memory that each procedure accesses. The compiler uses intraprocedural analysis algorithms

```
 1 void sort(int *d, int *t, int n) {
 2   int *d1, *d2, *d3, *d4, *d5;
 3   int *t1, *t2, *t3, *t4;
 4   if (n < CUTOFF) {
 5     insertionsort(d, d+n);
 6   } else {
 7     d1 = d; t1 = t;
 8     d2 = d1 + n/4; t2 = t1 + n/4;
 9     d3 = d2 + n/4; t3 = t2 + n/4;
10     d4 = d3 + n/4; t4 = t3 + n/4;
11     d5 = d4+(n-3*(n/4));
12
13     sort(d1, t1, n/4);
14     sort(d2, t2, n/4);
15     sort(d3, t3, n/4);
16     sort(d4, t4, n-3*(n/4));
17
18     merge(d1, d2, d2, d3, t1);
19     merge(d3, d4, d4, d5, t3);
20
21     merge(t1, t3, t3, t1+n, d);
22   }
23 }

24 void merge(int *l1, int *h1,
25            int *l2, int *h2, int *d) {
26   while ((l1 < h1) && (l2 < h2))
27     if (*l1 < *l2) *d++ = *l1++;
28     else *d++ = *l2++;
29   while (l1 < h1) *d++ = *l1++;
30   while (l2 < h2) *d++ = *l2++;
31 }

32 void insertionsort(int *l, int *h) {
33   int *p, *q, k;
34   for (p = l+1; p < h; p++) {
35     k = *p;
36     for (q = p-1; l <= q && k < *q; q--)
37       *(q+1) = *q;
38     *(q+1) = k;
39   }
40 }

41 void main() {
42   int n, *data, *temp;
43   scanf("%d", &n);
44   if (n > 0) {
45     data = (int*) malloc(n*sizeof(int));
46     temp = (int*) malloc(n*sizeof(int));
47     /* code to initialize the array */
48     sort(data, temp, n);
49     /* code that uses the sorted array */
50   }
51 }
```

Fig. 1. Divide and Conquer Sorting Example

to verify the design information, then uses the verified information to parallelize the program.

2 Example

Figure 1 presents a recursive, divide and conquer merge sort program. The `sort` procedure on line `1` takes an unsorted input array `d` of size `n`, and sorts it, using the array `t` (also of size `n`) as temporary storage. In the divide part of the algorithm, the `sort` procedure divides the two arrays into four sections and, in lines `13` through `16`, calls itself recursively to sort the sections. Once the sections have been sorted, the combine phase in lines `18` through `21` produces the final sorted array. It merges the first two sorted sections of the `d` array into the first half of the `t` array, then merges the last two sorted sections of `d` into the last half of `t`. It then merges the two halves of `t` back into `d`. The base case of the algorithm uses the insertion sort procedure in lines `32` through `40` to sort small sections.

```
merge(int *l1, int *h1,
      int *l2, int *h2,
      int *d) {
  context {
    input , output :
    l1 -> main:alloc1,
    h1 -> main:alloc1,
    l2 -> main:alloc1,
    h2 -> main:alloc1,
    d -> main:alloc2
  }
  context {
    input , output :
    l1 -> main:alloc2,
    h1 -> main:alloc2,
    l2 -> main:alloc2,
    h2 -> main:alloc2,
    d -> main:alloc1
  }
}
```

```
insertionsort(int *l, int *h) {
  context {
    input , output :
    l -> main:alloc1,
    h -> main:alloc1
  }
}

sort(int *d, int *t, int n) {
  context {
    input , output :
    d -> main:alloc1,
    t -> main:alloc2
  }
}
```

Fig. 2. Points-To Design Information

```
merge(int *l1, int *h1,
      int *l2, int *h2,
      int *d) {
  reads [l1,h1-1], [l2,h2-1];
  writes [d,d+(h1-l1)+(h2-l2)-1];
}
```

```
insertionsort(int *l, int *h) {
  reads and writes [l,h-1];
}

sort(int *d, int *t, int n) {
  reads and writes [d,d+n-1],
                   [t,t+n-1];
}
```

Fig. 3. Access Region Design Information

2.1 Design Information

There are two key pieces of design information in this computation: information about where each pointer variable points to during the computation, and information about the regions of the arrays that each procedure accesses. Our design language enables designers to express both of these pieces of information, enhancing the transparency of the code and enabling the parallelization transformation described below in Section 2.4.

Figure 2 shows how the designer specifies the pointer information in this example. For each procedure, the designer provides a set of *contexts*. Each context is a pair of *input* points-to edges and *output* points-to edges. The input set of edges represents the pointer aliasing information at the beginning of the procedure, and the output set of edges represents that information at the end of the procedure for that given input. Therefore, each context represents a partial transfer function: it describes the effect of the execution of the procedure for a given input points-to information. The pointer analysis takes place at the

granularity of *allocation blocks*. There is one allocation block for each static or dyanamic allocation site. The design information identifies each allocation site using the name of the enclosing procedure and a number that specifies the allocation site within the procedure. In our example, the input and the output information are the same for all contexts, which means that all procedures in our example have identity transfer functions.

Figure 3 shows how the designer specifies the accessed memory regions in the example. The regions are expressed using *memory region expressions* of the form $[l, h]$, which denotes the region of memory between l and h, inclusive. These regions are expressed symbolically in terms of the parameters of each procedure. This symbolic approach is required because during the course of a single computation, the procedure is called many times with many different parameter values.

As the example reflects, both pointer and access region specifications build on the designer's conception of the computation. The specification granularity matches the granularity of the logical decomposition of the program into procedures, with the specifications formalizing the designer's intuitive understanding of the regions of memory that each procedure accesses.

2.2 Pointer Design Conformance

The compiler verifies that the program conforms to its pointer design as follows. For each procedure and each context, the compiler performs an intraprocedural, flow-sensitive pointer analysis of the body of the procedure. At call sites, the compiler matches the current points-to information with the input information of one of the contexts of the callee procedure. It uses the output information from matched context to compute the points-to information after the call. If no matching context is found, the pointer design conformance fails. In our example, during the intraprocedural analysis of procedure `sort`, the compiler matches the current points-to information at lines `18` and `19` with the first context for `merge`, and the points-to information at line `21` with the second context of `merge`. The `sort` and `insertionsort` procedures each have only one context, and the compiler successfully matches the context at each call to one of these two procedures. Note that the pointer design information directly gives the fixed-point solution for the recursive procedure `sort`. The compiler only checks that this solution is a valid solution.

2.3 Access Region Design Verification

The compiler verifies the access region design in two steps. The first step is an intraprocedural analysis, called *bounds analysis*, that computes lower and upper bounds for each pointer and array index variable at each program point. This bounds information for variables immediately translates into the regions of memory that the procedure directly accesses via load and store instructions.

The second step is the verification step. To verify that for each procedure the design access regions correctly reflect the regions of memory that the whole

computation of the procedure accesses, the compiler checks the following two conditions:

1. the design access regions for each procedure include the regions directly accessed by the procedure, and
2. the design access regions for each procedure include the regions accessed by its invoked procedures.

In our example, for procedure `insertionsort` the compiler uses static analysis to compute the bounds of local pointer variables `p` and `q` at each program point, and then uses these bounds at each load and store in the procedure to detect that `insertionsort` directly reads and writes the memory region `[l,h-1]`. The compiler easily checks that this region is included in the access region for `insertionsort` from the design specification. The verification proceeds similarly for procedure `merge`.

For the recursive procedure `sort`, the design specifies two access regions: `[d,d+n-1]` and `[t,t+n-1]`. For the first recursive call to sort at line 13, the compiler uses the design access regions to derive the access regions for this particular call statement: `[d,d+n/4-1]` and `[t,t+n/4-1]`. It then verifies that these access regions are included in the design access regions for `sort`: $[\mathtt{d}, \mathtt{d} + \mathtt{n}/4 - 1] \subseteq [\mathtt{d}, \mathtt{d} + \mathtt{n} - 1]$ and $[\mathtt{t}, \mathtt{t} + \mathtt{n}/4 - 1] \subseteq [\mathtt{t}, \mathtt{t} + \mathtt{n} - 1]$. The compiler uses a similar reasoning to verify that all the call statements in `sort` comply with the design specification, and concludes that the implementation of `sort` conforms to its design.

A negative result in the verification step often indicates a subtle array addressing bug. For example, changing the `<` operator to `<=` in lines 26, 29, or 30 causes the compiler to report that the procedure does not conform to its design, as does changing `l+1` to `l` on line 34.

2.4 Parallelization

There are two sources of concurrency in the example: the four recursive calls to the `sort` procedure can execute in parallel, and the first two calls to the `merge` procedure can execute in parallel. Executing these calls in parallel leads to a recursively generated form of concurrency in which each parallel sort task, in turn, recursively generates additional parallel tasks.

The compiler recognizes this form of concurrency by comparing pairs of region expressions from different procedure calls and statements to determine if they are independent. Two region expressions are independent if they denote disjoint (non-overlapping) regions of memory or they both denote regions that are read. In our example, the four recursive calls to the `sort` procedure access independent region expressions and can execute in parallel with each other, as do the first two calls to the `merge` procedure.

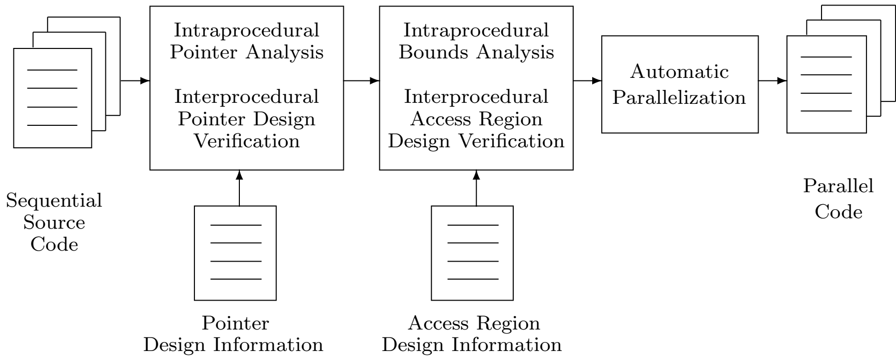

Fig. 4. The Structure of the Compiler

3 Structure of the Compiler

Figure 4 presents the general structure of the compiler. The compiler first uses a context-sensitive, flow-sensitive, and intraprocedural pointer analysis to verify the pointer design information [15]. It next uses an intraprocedural algorithm to verify the design access regions. Finally, it uses the verified design information to automatically parallelize the computation. We next discuss the static analysis and design verification algorithms in more detail.

3.1 Pointer Analysis and Design Verification

The intraprocedural pointer analysis extracts points-to information at each program point[1]. It represents points-to information using points-to graphs [5]. The nodes in this graphs are program variables, and the edges in the graph represent points-to relations between variables. The compiler also handles dynamically allocated objects, distinguishing between them based on their allocation site. The compiler uses special variables, called ghost variables, to represent variables on the stack that are not in the scope of the currently analyzed procedure, but are accessible via parameters or global pointers from the current procedure.

In the intraprocedural analysis, our compiler uses a flow-sensitive, pointer analysis algorithm [15]. It uses a standard dataflow analysis approach, with specific analysis rules for pointer assignments via copy, load, and store statements. It uses the design information to compute transfer functions for procedure calls. Each context in the design specifies a partial transfer function for that procedure [18], and the analysis directly uses the output of the context whose input matches the current points-to information at the call.

In the specifications, points-to graphs are represented as sets of edges between program variables. Dynamically allocated objects are specified using both

[1] Although in this section we present how we verify points-to information, similar techniques can be employed for any other dataflow information.

the name of the enclosing procedure that allocates them and a number indicating which allocation site within that procedure it is referring to. For instance `main:alloc1` represents an object allocated at the first dynamic allocation site in procedure `main`. Ghost variables are specified using their type, for instance `ghost(int[10])` describes an array of integers allocated on stack, accessible, but not visible to the current procedure.

When the analysis of each context completes, the compiler checks that the analysis result for that context matches the corresponding output points-to information from the design. If the design is verified, the pointer analysis results at each program point can be safely used in the following stages of the compiler.

3.2 Access Region Analysis and Design Verification

The compiler next uses the same approach of combining static analysis and design verification to derive symbolic regions of memory that the whole computation of each procedure accesses. The lower and upper bounds of these regions are symbolic polynomial expressions in the initial values of the parameters of the enclosing procedure. Both the analysis and the design verification presented in this section separately keep track of read and written regions of memory.

The symbolic access region in the specification represent regions within allocation blocks. The general format of an access region within an allocation block relative to a procedure f is: $[\,p : l\,,\,h\,]$ where l and h are symbolic expressions in the initial parameters of f and p is a pointer variable. This denotes a region with lower bound l and upper bound h within all the allocation blocks pointed to by p, at the beginning of f, for all points-to contexts of f. If the lower bound l is a simple symbolic expression consisting of a single term equal to p, then we use the shortcut notation $[\,l\,,\,h\,]$. All the specifications in our example from Figure 1 use this shortcut notation.

The compiler extracts and verifies the access region information as follows:

1. **Static Analysis:** The compiler first performs an intraprocedural analysis, called *bounds analysis*, to extract lower and upper bounds for each pointer and array index variable at each program point. The algorithm is presented in detail in [16]. The compiler then uses the extracted bounds information for pointers and array indices to derive access regions for each memory access (i.e. each load and store) in the program. The compiler finally combines these access regions for loads and stores and derives the regions of memory that each procedure *directly* accesses.
2. **Design Verification:** The compiler next uses the intraprocedural access region results to verify that the access region design specification correctly characterizes the memory regions that the whole execution of each procedure accesses. To verify the safety of the design access regions, the compiler checks two conditions. First, the design access regions of each procedure should include the access regions directly accessed by that procedure. Second, the design access regions of each procedure should include the design access regions of all its invoked procedures. If both conditions hold for all procedures, the design is verifies. Otherwise, access design verification fails.

During the verification process at call statements, the compiler uses the access regions of the callee to derive an access region for the call statement. But the analysis of the callee produces a result in terms of the initial values of the callee's parameters. The result for the caller must be expressed in terms of the caller parameters, not the callee parameters. The *symbolic unmapping* algorithm performs this change of analysis domain. A detailed description and a formal definition of symbolic unmapping is given in [16]. The idea behind symbolic umapping of an access region is to replace the callee's parameters in the region bounds with the actual parameters at the call site, and then use the bounds information at the call site to express the access region in terms of the initial values of the caller's parameters.

Once it verifies the access region design information, the compiler can safely use it to detect sequences of independent calls and generate parallel code to execute them concurrently.

4 Experimental Results

We have implemented a compiler that combines the static analysis algorithms and the design verification algorithms presented in this paper. This compiler was implemented using the SUIF compiler infrastructure [1]. We implemented all of the analyses, including the pointer analysis, from scratch starting with the standard SUIF distribution. Our compiler generates parallel code in Cilk [7], a parallel dialect of C.

We present experimental results for two recursive sorting programs (Quicksort and Mergesort), a divide and conquer blocked matrix multiply (BlockMul), a divide and conquer LU decomposition (LU), and a scientific computation (Heat). We would like to emphasize the challenging nature of the programs in this benchmark set. Most of them contain multiple mutually recursive procedures, and have been heavily optimized by hand to extract the maximum performance. As a result, they heavily use low-level C features such as pointer arithmetic and casts.

4.1 Design Conformance

Using the approach presented on this paper, the compiler successfully verified that all the benchmarks comply to both their pointer design and to their access region design. The compiler used the extracted intraprocedural pointer information and access region information to carry out the design verification process.

4.2 Design Information Size and Complexity

We compare the complexity of the access region design as opposed to the program by computing the ratio of the number of bytes in the program divided by the number of bytes in the design. Table 1 separately presents the results for pointer design specifications and access region design specifications. which show that our set of benchmark programs is between 6 and 27 times larger than their designs.

Table 1. Ratio of Program Size to Design Size

Program	Program to Pointer Design Ratio	Program to Region Access Design Ratio
Quicksort	10	15
Mergesort	6	14
Heat	10	12
BlockMul	15	27
LU	11	12

Table 2. Pointer Analysis Running Times (in seconds)

	Pointer Analysis Alone	Pointer Analysis and Design Information
Quicksort	0.02	0.01
Mergesort	0.05	0.04
Heat	0.13	0.09
BlockMul	3.45	1.84
LU	0.30	0.15

Our own qualitative assessment of the design information is that it is very easy for the designer to provide, in part because it is a natural, intuitive extension of the procedure interface, and in part because it is so small in comparison with the programs.

4.3 Compiler Complexity and Efficiency

Table 2 shows the running times of the pointer analysis phase using combined program analysis and design information compared to program analysis alone. The availability of design information can produce speedups up to a factor of 2 for our set of benchmarks. For the access region phase the running times were roughly the same with and without design information. Here the bottleneck was the intraprocedural analysis, which is executed in both cases.

The availability of design information significantly decreased both the complexity and the implementation time of the analysis. Compared to the implementation in our previous work for the automatic parallelization of divide and conquer algorithms [14,16], the design-based approach presented in this paper eliminated sophisticated interprocedural algorithms based on fixed-point algorithms or on reductions to linear programs. These complex analyses were replaced by the simple design verification algorithms presented in the current paper. This reduction in compiler complexity also translated in a reduction of implementation time from the order of months to the order of days for the replaced sections of the compiler.

Table 3. Absolute Speedups

Programs	Number of Processors				
	1	2	4	6	8
Quicksort	1.00	1.99	3.89	5.68	7.36
Mergesort	1.00	2.00	3.90	5.70	7.41
Heat	1.03	2.02	3.89	5.53	6.83
BlockMul	0.97	1.86	3.84	5.70	7.54
LU	0.98	1.95	3.89	5.66	7.39

4.4 Automatic Parallelization

Our analysis was able to automatically parallelize all of the applications. We ran the benchmarks on an eight processor Sun Ultra Enterprise Server. Table 3 presents the speedups. These speedups are given with respect to the sequential versions, which execute with no Cilk overhead. For Heat, the Cilk program running on one processor runs faster than the sequential version, in which case the absolute speedup is above one for one processor. We ran Quicksort and Mergesort on a randomly generated file of 8000000 numbers and BlockMul and LU on a 1024 by 1024 matrix.

5 Related Work

5.1 Access Specifications

The concept of allowing programmers to specify how constructs access data is a continually arising subtheme in programming languages. The effect system in FX/87, for example, allows programmers to specify the *effects* of each procedure, i.e., the regions that it accesses [8]. The type checking algorithm is extended to statically verify that the specified effects correctly reflect the accesses of the procedure. Access declarations in Jade allow programmers to specify how tasks access shared objects [13]. The access declarations are used to parallelize the program, and are dynamically checked by the Jade run-time system. In both Jade and FX/87, the specifications operate at the granularity of complete objects — there is no way to specify that a procedure or task accesses part of an array or object. The access specifications in this paper, on the other hand, operate at the granularity of subregions of the accessed arrays. They therefore enable the compiler to recognize (and parallelize) procedure calls that access disjoint regions of the same array.

5.2 Interprocedural Array Region Analysis

Several researchers have developed systems that automatically characterize the array regions that procedures access. The first systems were designed to analyze

scientific programs with loop nests that manipulate dense matrices using affine access functions [17,12,11]. These systems use the loop bounds and the array index expressions to derive the array regions that each procedure accesses. They then propagate accessed array regions from callees to callers to derive the regions accessed by the complete execution of each procedure. Researchers have recently generalized this approach for recursive procedures that access data via pointers [14,9]. An issue is maintaining precision in the face of the fixed-point computations used to analyze recursive procedures. Our recent generalization of the intraprocedural approach presented in Section 3 to accurately analyze recursive procedures without fixed-points eliminates this particular problem [16].

The bottom line is that it is possible, in principle, to attack the problem of parallelizing divide and conquer programs without design information in the form of access regions. We nevertheless see such design information and design conformance as playing a desirable role in this context, for the following reasons:

- **Simplicity:** Access regions enable the compiler to apply a simple intraprocedural algorithm. Eliminating the interprocedural analysis significantly simplifies the structure of the compiler and its analysis algorithms. Improvements include increased confidence in the correctness of the compiler and a reduction in the implementation time and complexity.
- **Independence:** Access regions enable the compiler to analyze and compile each procedure independently of all other procedures. The analysis is therefore more efficient and scalable since it does not have to perform a global analysis. The design information also enables the compiler to support separate compilation, unanalyzable libraries, and missing code in programs under development.
- **Development Improvements:** Access regions are an intuitive formalization of a key aspect of the design of the program. They are easy for designers to provide, in part because they simply crystalize information that the designer must already have available to successfully design the algorithm, and in part because the designer provides only a small amount of information at procedure boundaries. They also provide a natural extension to standard procedure interfaces, improving the transparency of the code and giving clients additional information about the interface of the procedure. Finally, they can help the debugging process: subtle array addressing bugs often show up as violations of the declared access regions.

6 Future Work

Information about the ranges of pointer and array index variables can be used for purposes other than automatic parallelization. For example, many security problems are caused by incorrect programs that an attacker can coerce into violating its array bounds. We believe that enabling the designer to explicitly state the array referencing expectations inherent in the design would help developers produce software without these problems. Developers could therefore use our

approach to verify that the program has no security vulnerabilities caused by array bounds violations.

Languages such as Java use dynamic checks to eliminate array bounds violations. The advantage is that array bounds violations are caught before they corrupt the system; the disadvantage is the overhead of performing the array bounds checks dynamically. And an array bounds violation is still an error, and typically causes the program to fail. By statically verifying that programs do not violate their array bounds, our proposed techniques can both eliminate dynamic array bounds check overhead and improve the reliability of the delivered software.

7 Conclusion

This paper presents design-driven compilation, a technique for using design information to improve the analysis and compilation of the program. Design-driven compilation uses design information to drive its analysis and verify that the program conforms to its design. The main advantages of design-driven compilation are the fidelity to the designer's expectations, analysis modularity, and simplicity and efficiency of the compiler. We have applied this approach to the problem of automatic parallelization of divide and conquer programs. Our results show that the design information is small compared to the program, works well with the designer's intuitive conception of the structure, decreases the complexity of the compiler while increasing its efficiency, and enables the compiler to generate parallel code with excellent performance.

In the future, we anticipate that design conformance will become an increasingly important. In addition to enabling the compiler to better analyze both complete and incomplete programs, it will also help designers and implementors deliver more reliable programs that are guaranteed to conform to their designs. We anticipate that this automatically checked connection between the design and the implementation will significantly increase the role that formal designs play during the implementation and maintainance phases, reducing the cost of these phases and increasing the robustness of the delivered software.

Acknowledgements. We would like to thank Daniel Jackson for many interesting conversations regarding design conformance.

References

1. S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
2. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995. ACM, New York.

3. S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint Malo, France, June 1999.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introductions to Algorithms.* The MIT Press, Cambridge, Mass., Cambridge, MA, 1990.
5. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.
6. J. Frens and D. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.
7. M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.
8. D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1987.
9. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. Technical report, IBM T. J. Watson Research Center, 1999.
10. F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.
11. M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press, Los Alamitos, Calif.
12. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
13. M. Rinard and M. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.
14. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
15. R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.
16. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
17. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
18. R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.