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 requires the programmer to formally specify some additional design information, it offers a range of benefits, such as: *fidelity* to the designer's high-level expectations, early and automatic detection of design nonconformance bugs; *analysis modularity*, with support for local analysis, separate compilation, and library support; and *simplicity* and *efficiency* of the compiler. We consider that the appropriate abstraction level for design information is the procedure level, with design specifications expressed as procedure interfaces. 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.

This paper reports our experience using design-driven compilation to automatically parallelize divide and conquer programs. The designer provides a design specification for pointer information, in the form of partial transfer functions, as well as a design specification for the regions of memory that each procedure accesses. The compiler first verifies the two pieces of design information then uses these specifications to detect the procedures that can execute in parallel. Our experimental results show that this approach enables the compiler to automatically parallelize a range of divide and conquer programs, that the complexity of the compiler decreases while its efficiency increases, and that, compared to the program, the design information is small, intuitive, and easy to provide.

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. In a design-driven compiler, the designer provides design information about the program. The compiler then uses this information to drive its analysis and verify that the program conforms to its design.

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

1.1 Design Conformance

The design information of a program provides properties that the program is intended to fulfill. This information is implicitly encoded in the program, and the compiler should otherwise need sophisticated program analysis techniques to extract it. The explicit (and redundant) specification of design information however offers a range of advantages. First, it clearly states the programmer's intent, and nonconformance 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 us-

*This research was supported in part by DARPA Contract F3615-00-C-1692, NSF Grant CCR-0086154, and NSF Grant CCR-9702297.

ing *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 examples of procedure interfaces. We use procedure interfaces for pointer information, to express how procedures change the points-to information. We also use procedure interfaces to express the regions of memory that the whole computation of each procedure accesses.

At the intraprocedural level, it is not practical to expect the programmer provide design information at each program point. The design specification could provide at most flow-insensitive information that conservatively approximates the information at all program points. For instance, the design information could express flow-insensitive pointer information for each procedure. But since program-point information and other low-level properties are crucial for some compiler transformations or verifications, the key to the success of our approach is to *combine design specifications with program analysis*. Design information specifies high-level properties of the program, and static analysis extracts the low-level properties. The compiler verifies the design information using the extracted low-level analysis result. The compiler further uses both the verified and the extracted information for subsequent transformations and verifications.

To summarize, we believe that the appropriate abstraction level for design specification is the procedure level, and that design specifications for procedures should be used along with powerful program analyses that extract information at the intra-procedural level. 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:* The design information enables the early and automatic detection of design non-conformance bugs. The compiler can for instance automatically detect subtle errors (such as off-by-one errors) in array index and pointer arithmetic calculations.
   - *Enhanced Design Utility:* Our approach can verify 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.
   - *Library Support:* The design information enables the compiler 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:**
   - *Simple and Efficient Compiler:* 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 specifications. Improvements include increased confidence in the correctness of the compiler, a reduction in the implementation time and complexity, and an increased compiler efficiency.

1.2 Divide and Conquer Programs

We have applied design-driven compilation to a challenging problem: the automatic parallelization of divide and conquer programs. These programs solve problems by breaking them up into smaller subproblems, recursively solving the subproblems, then combining the results to generate a solution to the original problem. A simple algorithm that works well for small problem sizes terminates the recursion. Good divide and conquer algorithms exist for a large variety of problems, including sorting, matrix manipulation, and many dynamic programming problems [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].

1.3 Automatic Parallelization

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. The compiler therefore

has to solve two main problems for the automatic parallelization of such applications: pointer analysis and analysis of memory regions accessed by procedures.

Our solution to this problem is to enable the designer to provide specifications about these two kinds of information. First, design properties for pointer information are given in the form of procedure transfer functions: for each procedure, they specify a number of input contexts as input points-to graphs, and the associated output points-to graphs. The compiler uses an intra-procedural pointer analysis to verify the validity of these transfer functions. Second, the designer provides information about the regions of memory that the whole computation of each procedure accesses. Here the compiler performs an intra-procedural analysis to extract the regions of memory that each procedure directly accesses. It then uses the design information to derive the regions accessed by the procedures that it invokes. Finally, it checks that the derived regions are included in the design regions. In effect, in both cases the compiler uses the design information as an induction hypothesis to increase the reach of the analysis.

Once the compiler knows the memory regions accessed by each procedure, it can compare memory regions to determine if different procedure calls can legally execute in parallel.

1.4 Contributions

This paper makes the following contributions:

- **Design Driven Compilation:** It introduces design-driven compilation, where the compiler uses design information to drive its analysis and verify that the program conforms to its design. It also proposes the procedural level as the natural abstraction level for design specifications. The design-driven approach offers a range of benefits, including fidelity to the programmer's expectations, analysis modularity, and compiler simplicity and efficiency.

- **Applications:** It shows how to apply the concept of design-driven compilation to a compiler for automatically parallelizing divide and conquer algorithms. It shows how to use the design-driven compilation approach in two phases of the compiler, to specify pointer design information, and access region design information. The key to the success of this effort is the combination of an intuitive design language that enables the designer to easily express the relevant design information with a powerful static analysis that handles the low-level details of verifying that the program conforms to its design.

- **Experimental Results:** It presents experimental results from an implemented compiler for divide and conquer programs. These results show that the compiler is able to verify the conformance of the program to its design and that the resulting automatically parallelized programs exhibit performance comparable to the corresponding manually parallelized versions. They also show that the size and complexity of the design information is small compared to the program.

The remainder of the paper is organized as follows. In Section 2 we present an example that illustrates how our compiler applies design conformance to automatically parallelize a divide and conquer sort program. Section 3 presents the analysis algorithm. Section 4 presents experimental results from our compiler, which uses design-driven compilation to automatically parallelize divide and conquer algorithms. Sections 5 and 6 discuss related and future work. We conclude in Section 7.

2 Example

Figure 1 presents a recursive, divide and conquer merge sort program. The `sort` procedure on line `18` takes an unsorted input array `d` of size `n`, and sorts it, using the array `t` (also of size `n`) as temporary storage. The algorithm is structured as follows.

In the divide part of the algorithm, the `sort` procedure divides the two arrays into four sections and, in lines `29` through `32`, calls itself recursively to sort the sections. Once the sections have been sorted, the combine phase in lines `34` through `37` 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 `9` through `17` to sort small sections.

As is often the case with divide and conquer programs, the `sort` program identifies subproblems using pointers into dynamically allocated memory blocks that hold the data. It accesses these blocks via these pointers. This strategy leads to code containing significant amounts of pointer arithmetic and pointer comparison operators. Note, for example, the pointer arithmetic in lines `24` through `28` and the `<` pointer comparison operators in lines `3`, `6`, and `7`.

2.1 Design Information

There are two key pieces of design information in this computation: information about where each pointer variables points to during the computation, and information about the regions of the arrays that each procedure accesses. Our design language enables programmers 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 programmer 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 the pointer 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. 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. Also, dynamic memory allocations are specified using both the name of the enclosing procedure and a number indicating the occurence of the dynamic allocation site within that procedure.

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,

```
1:  void merge(int *l1, int *h1,
2:             int *l2, int *h2, int *d) {
3:    while ((l1 < h1) && (l2 < h2))
4:      if (*l1 < *l2) *d++ = *l1++;
5:      else *d++ = *l2++;
6:    while (l1 < h1) *d++ = *l1++;
7:    while (l2 < h2) *d++ = *l2++;
8:  }

9:  void insertionsort(int *l, int *h) {
10:   int *p, *q, k;
11:   for (p = l+1; p < h; p++) {
12:     k = *p;
13:     for (q = p-1; l <= q && k < *q; q--)
14:       *(q+1) = *q;
15:     *(q+1) = k;
16:   }
17: }

18: void sort(int *d, int *t, int n) {
19:   int *d1, *d2, *d3, *d4, *d5,
20:       *t1, *t2, *t3, *t4;
21:   if (n < CUTOFF) {
22:     insertionsort(d, d+n);
23:   } else {
24:     d1 = d; t1 = t;
25:     d2 = d1 + n/4; t2 = t1 + n/4;
26:     d3 = d2 + n/4; t3 = t2 + n/4;
27:     d4 = d3 + n/4; t4 = t3 + n/4;
28:     d5 = d4+(n-3*(n/4));

29:     sort(d1, t1, n/4);
30:     sort(d2, t2, n/4);
31:     sort(d3, t3, n/4);
32:     sort(d4, t4, n-3*(n/4));
33:
34:     merge(d1, d2, d2, d3, t1);
35:     merge(d3, d4, d4, d5, t3);
36:
37:     merge(t1, t3, t3, t1+n, d);
38:   }
39: }

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

Figure 1: Divide and Conquer Sorting Example

```
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
  }
}
```

Figure 2: Points-To Design Information

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 intra-procedural flow-sensitive pointer analysis of the body of the procedure. At each pointer assignment statement, new points-to edges are created. At call sites, the compiler tries to match the current points-to information with the input information of one of the contexts of the callee procedure. If no matching context is found, the pointer design conformance fails. In our example, during the intra-procedural analysis of procedure `sort`, the compiler matches the current points-to information at line 34 with the first context for `merge`, and the points-to information at line 37 with the second context of `merge`.

The compiler also matches the points-to information at the recursive calls of `sort` at lines 29, 30, 31, and 32 with the currently analyzed context. Note that the pointer design information directly gives the fixed-point solution for the recursive procedure `sort`. The compiler only checks that

```
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];
}
```

Figure 3: Access Region Design Information

this is a valid fixed-point solution.

2.3 Access Region Design Verification

The compiler verifies the access region design in two steps. The first step is an intra-procedural 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 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, the compiler checks the following two conditions:

- the design access regions for each procedure should include the regions directly accessed by the procedure;
- the design access regions for each procedure should include the regions accessed by its invoked procedures;

Note here that for a given call statement, the symbolic regions are expressed in terms of the formal parameters of the invoked procedure, not the formal parameters of the analyzed procedure. As described in Section 3, the algorithm uses the symbolic bounds of the actual parameters at call sites to translate the region expressions of invoked procedure in terms of formal parameters of the analyzed procedure.

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` 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 29, 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.

```
29:     spawn sort(d1, t1, n/4);
30:     spawn sort(d2, t2, n/4);
31:     spawn sort(d3, t3, n/4);
32:     spawn sort(d4, t4, n-3*(n/4));
33:     sync ;
34:     spawn merge(d1, d2, d2, d3, t1);
35:     spawn merge(d3, d4, d4, d5, t3);
36:     sync ;
37:     merge(t1, t3, t3, t1+n, d);
```

Figure 4: Parallel Code Fragment for Sorting Example

A negative result in the verification step often indicates a subtle array addressing bug. For example, changing the `<` operator to `<=` in lines `3`, `4`, or `7` causes the compiler to report that the procedure does not conform to its design, as does changing `l+1` to `l` on line `11`.

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 compiler uses pointer analysis information to determine that $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{t}+\mathtt{n}/4, \mathtt{t}+\mathtt{n}/2-1]$ denote memory regions in different allocation blocks and are therefore independent. It uses a symbolic comparison operation to determine that $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{d}+\mathtt{n}/4, \mathtt{d}+\mathtt{n}/2-1]$ denote nonoverlapping regions of the same block. It can use similar strategies to determine that all of the other pairs are independent, and that the calls can execute in parallel.

Our compiler generates parallel code in Cilk [7], a parallel dialect of C. In Cilk, a `spawn` construct in front of a procedure call instructs the compiler that the designer expects the call to execute in parallel; the `sync` construct indicates that previous parallel calls may conflict with the computation after the `sync` statement, and that the compiler should generate code that blocks at the `sync` statement until all of these parallel calls have finished their execution. Figure 4 presents the generated parallel code for recursive procedure `sort` in our example.

3 Structure of the Compiler

Figure 5 presents the general structure of the compiler, which combines the code and the design information to perform the following three phase:

1. **Pointer Analysis and Design Verification:** We use an intraprocedural, context-sensitive, flow-sensitive

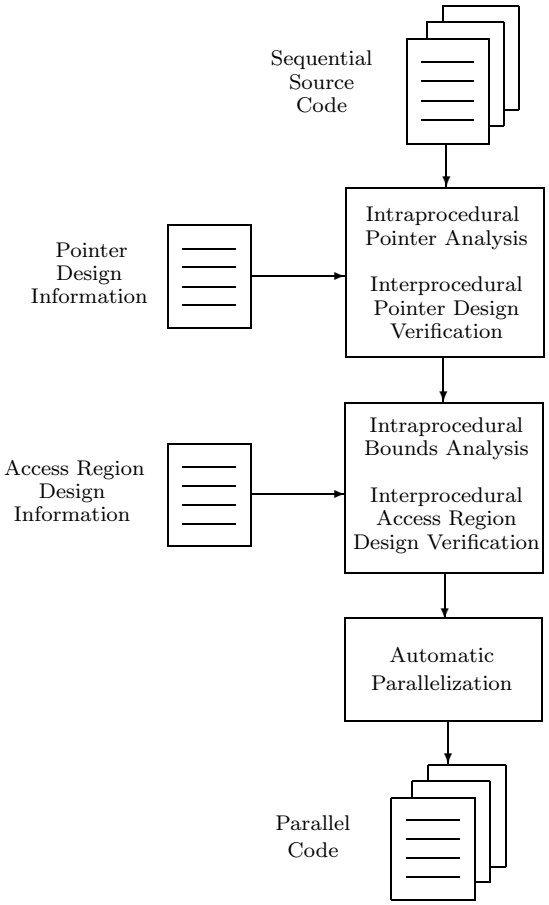

Figure 5: The Structure of the Compiler

pointer analysis algorithm to determine the allocation blocks accessed via pointer variables [15]. At each call statement, the analysis uses the procedure transfer functions from the design specification to compute the pointer information after the execution of the statement. The compiler finally verifies that, for each procedure and each context, the pointer information at the end of the procedure complies the design specification.

This phase provides memory disambiguation at the granularity of *allocation blocks*. There is an allocation block per static or dynamic allocation site. All of the elements of each array are merged together to be represented by the allocation block from the array's allocation site. The next phase of the compiler refines this coarse grain memory access information with access regions within the arrays.

2. **Access Region Analysis and Design Verification:** This phase first performs a static analysis to determine the regions of memory that procedures directly access. This analysis derives symbolic lower and upper bounds for each pointer and array index variable at each program point. These bounds directly translate into access regions for pointer-based or index-based memory accesses, enabling the compiler to derive the access regions directly accessed by each procedure.

   The compiler then verifies the correctness of the access region design, which specify the regions that the whole execution of each procedure access. To check the design, the compiler first verifies that the regions directly accessed by each procedure are included in the design access regions. It then verifies that at each call statement, the regions accessed by the caller procedure at that specific call site are included in the design access regions of the caller procedure.

3. **Parallelization:** The compiler uses the verified design information to determine sequences of independent procedure calls, and then generates code to execute them concurrently.

We next present the static analysis and design verification algorithms in more detail, emphasizing the verification phases.

3.1 Pointer Analysis and Design Verification

In this stage, the compiler extracts intra-procedural pointer information at each program point, and verifies the inter-procedural pointer information given in the design specification[1].

Our compiler represents pointer 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, and distinguishes 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.

At the intraprocedural analysis, our compiler uses a flow-sensitive, context-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. At the interprocedural level, our approach is context-sensitive: it distinguishes between invocations of the same procedure with different input pointer information. The design specification provides procedure interfaces that describe the partial transfer functions for procedures [18]. A partial transfer function consists of a pair of an input points-to graph, and an asociated output points-to graph, which describes the effects of the execution of the procedure for that particular input. For each procedure there may be several partial transfer functions.

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

The combined analysis and verification algorithm is straightforard. It comprises two subphases:

1. **Static Analysis and Use of Design Information:** For each procedure and each context in the design

[1] Although in this section we present how we combine static analysis and design verification for pointer information, similar techniques can be employed for any other dataflow information.

specification, the compiler performs an intra-procedural dataflow analysis of the procedure body. At each call site, the compiler replaces the variables not visible to the callee with ghost variables, through a mapping process [5]. The compiler then tries to match the mapped points-to graph to the input points-to graph of one of the contexts in the design specification. If no matching points-to graph is found, the design verification fails. If a context with matching input graph exists, the compiler uses the output points-to graph of the matching context. The compiler then unmaps the the ghost variables back to their associated variables in the caller. After the unmapping, the compiler resumes the analysis of the caller at the program point after the call statement.

2. **Design Verification:** The compiler then checks, for each procedure and each context, if the analysis result at the end of the procedure matches the output pointer information of the context. If so, the pointer design specification is verified. If the design ouput for some context does not match the analysis result, the verification fails.

Let's consider the analysis and design verification for procedure `sort` in our example from Figure 1. The compiler first analyzes the only context this procedure. It starts the analysis with a points to information where its pointer parameters `d` and `t` point to the dynamically allocated arrays at lines `45` and `46`, respectively. During the analysis of this procedure, the compiler creates points-to edges from the local variables of this procedure to these arrays. When the analysis reaches the call statements, the compiler matches the context of `sort` at all the recursive calls, matches the first two calls to procedure `merge` with the first context of this procedure, and matches the last call to `merge` with its second context. In all these cases, the compiler uses the design specification for the callee procedure to detect that the points-to information does not change after the execution of any of these calls. Finally, the compiler verifies that the analysis result at the end of procedure `sort` matches the output from the design specification. The compiler thus concludes that procedure `sort` conforms to its pointer design.

If the pointer design is verified, the pointer analysis results at each program point can be safely used in the following stages of the compiler. This pointer information provides information about the accessed pieces of memory at the granularity of allocation blocks, but does not distinguish between different regions in the same array. The next phase refines the memory access information with such access regions.

3.2 Access Region Analysis and Design Verification

In this phase, the compiler uses the same approach of combining static analysis and design verifications 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.

As in the case of pointer information, the compiler extracts and verifies the access region information in three phases:

1. **Static Analysis:** The compiler first performs an intra-procedural 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]. This algorithm is formulates the problem as a constraint system of symbolic inequalities. It then reduces this symbolic constraint system to a linear program. The solution to the linear program translates directly into lower and upper bounds for variables at each program point. 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.

Let's consider again our example from Figure 1. For `merge` and `insertionsort`, the bounds analysis computes direct access regions identical to the design access regions. For procedures `sort` and `main`, the analysis computes empty access regions. Therefore, in all these cases the first verification condition is fulfilled: the design access regions include the computed directly accessed regions. The compiler then checks all the call sites to ensure that the design access regions of the caller include the access region of the call

statement. Consider the recursive call `merge(d2,t2,n/4)` at line 30. The interface for `sort` specifies that this procedure accesses the regions `[d,d+n-1]` and `[t,t+n-1]`. Using symbolic unmapping, the compiler derives the following access regions for the recursive call at line 30: `[d+n/4;d+2*(n/2)-1]` and `[t+n/4;t+2*(n/2)-1]`. Using simple symbolic comparisons, the compiler checks that these regions are included in the design access regions of `sort`, `[d,d+n-1]` and `[t,t+n-1]` respectively. All call statements in the program are similarly verified. The compiler hence concludes that the program conforms to its access region design.

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. We present experimental results for several benchmark divide and conquer programs.

- **Quicksort:** Divide and conquer quicksort, uses insertionsort to terminate the recursion and sort small files.
- **Mergesort:** Divide and conquer mergesort, uses insertionsort to terminate the recursion and sort small files.
- **Heat:** Solves heat diffusion on a mesh.
- **BlockMul:** Divide and conquer blocked matrix multiply.
- **LU:** Divide and conquer LU decomposition.

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 were compliant 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.

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.

	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 1: Ratio of Program Size to Design Size

	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

Table 2: Pointer Analysis Running Times (in seconds)

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.

Even in the cases where the running times were not significantly dropped by the availability of design information, both compiler complexity and implementation time were significantly decreased. 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.

4.4 Automatic Parallelization

Our analysis was able to automatically parallelize all of the applications. We ran the benchmarks on an eight processor

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

Table 3: Absolute Speedups

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. In some cases 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 MatMul, NoTempMul, and LU on a 1024 by 1024 matrix.

5 Related Work

In this section we review two areas of related work: research on access specification languages, and research on interprocedural array region analysis.

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.