JuliusC: A Practical Approach for the Analysis of Divide-And-Conquer Algorithms

Paolo D'Alberto and Alexandru Nicolau

Information and Computer Science, University of California at Irvine
{paolo,nicolau}@ics.uci.edu *

Abstract. The development of divide and conquer (D&C) algorithms for matrix computations has led to the widespread use of high-performance scientific applications and libraries.
In turn, D&C algorithms can be implemented using loop nests or recursion. Recursion is extremely appealing because it is an intuitive means for the deployment of top-down techniques, which exploit data locality and parallelism naturally. However, recursion has been considered impractical for high-performance codes, mostly because of the inherent overhead of the division process into small subproblems.
In this work, we develop techniques to model the behavior of recursive algorithms in a way suitable for use by a compiler in estimating and reducing the division process overheads. We describe these techniques and JuliusC, a (lite) C compiler, which we developed to exploit them. JuliusC unfolds the application call graph (partially) and extracts the relations among function calls. As a final result, it produces a directed acyclic graph (DAG) modeling the function calls concisely. The approach is a combination of compile-time and run-time analysis and both have negligible complexity.
We illustrate the applicability of our approach by studying 6 test cases. We present the analysis results and we show how our (optimizing) compiler can use these results to increase the efficiency of the division process between 14 to 20 million times, for our codes.

1 Introduction

With the introduction of divide and conquer (**D&C**) algorithms for the solution of matrix computations (e.g., BLAS level 3 [27]), the performance of scientific applications has improved markedly (e.g., Lapack [1]). In practice, D&C computations exploit both spatial and temporal data locality and this data locality results in efficient use of resources in modern uniprocessor and multiprocessor systems. Due to this efficient use of resources, D&C computations can achieve extremely good performance.

D&C algorithms can be *a priori* implemented using either of two language constructs: loop nests or recursion. In this work, we refer to the algorithms

* This work has been supported in part by NSF Contract Number ACI 0204028 and by PBK Alumni in Southern California

implemented using loop nests as **blocked** [11, 2], and to those using recursion as **recursive** algorithms [15, 30].

Due to the importance of matrix algorithms, practical and efficient solutions of basic matrix computations have been the center of substantial efforts in terms of time (and money) in the last decades. As a result, ready-to-use and highly-tuned libraries for several systems have been proposed and used (e.g., Lapack [1]) and, more recently, self-installing and self-tuning libraries arose as new standards [37, 6] delivering astonishingly good performance. Briefly, self-installing and self-tuning libraries extract the parameters of an architecture by profiling; then they tailor the source code to those parameters and, eventually, they tune the application using trial-and-error techniques. These libraries are based on blocked algorithms (rather than recursive ones) because of the two recursive algorithm problems, which we shall introduce shortly, and for a number of other reasons: code legacy (e.g., FORTRAN has no-recursion), small code size (amenability to traditional aggressive compiler optimizations), negligible overhead and the ability to exploit locality by loop tiling [25].

Recursive algorithms are appealing because of their intuitive formulation and implementation. In fact, they are top-down solutions; they exploit data locality naturally at every level of cache in architectures with deep memory hierarchies [23]; they may present data dependency intuitively and, in turn, they present clear examples for the application of programming constructs driving the practical parallelization along the recursive calls naturally [34]. It is matter of fact, most recursive matrix algorithms, such as matrix multiply and LU-factorization, are *optimal cache oblivious algorithms* [17, 35]. That is, they have asymptotically optimal cache reuse at any cache level (without any tuning except for the register file, where some code tuning may further increase performance [8]). Ultimately, recursive algorithms represent an efficient solution using a higher level of abstraction, therefore a more intuitive format, freeing the developer from the details of the architecture on which these applications will run. Despite the above advantages, recursive algorithms have been considered impractical (for high-performance applications) for two reasons.

First, a compiler has a difficult time optimizing at which point the recursion should stop (**leaf computation**). Thus, if only recursion is applied, the leaf computation may have too few instructions leaving little room for optimizations, exploiting little register reuse, and, especially, incurring a high overhead due to the (otherwise avoidable) proliferation of small recursive procedure calls. Otherwise if recursion stops too soon, the leaf computation may be too large to exploit fully cache locality.

Second, any recursive algorithm *inherits* an overhead due to the **division process**, or partitioning, (intrinsic in D&C algorithm) into small subproblems. This overhead is proportional to the number of function-recursion calls and to the work involved in computing the actuals of a function call.

Several authors target the solution of the recursive-algorithm problems proposing applications based on both loop nests and recursion [19, 8, 16]. The division process is implemented by a recursive algorithm, exploiting the *oblivious* data cache locality. The leaf computation is implemented by a blocked algorithm, exploiting aggressive compiler optimizations. These authors propose to stop the recursion at a certain level and then call high-performance non-recursive routines. This is also known as **pruning**. Pruning reduces the number of function calls, and therefore it reduces the overhead (but it does not reduce the work per function call; that is, the computation of the actuals for every function call).

Note that with the explosive increase in the complexity of systems due to the rapid advances in semiconductor technology, the number of parameters describing a system also increases (e.g., several level of caches). This undermines the portability and the efficiency of current self-tuning high-performance applications because the complexity of probing the parameters, as well as of tailoring the source code to an architecture, will become impractical - if not quixotic. In contrast, recursive algorithms assure optimal locality and parallelism with little knowledge of the architecture, therefore recursive algorithms for matrix computations should become increasingly appealing, if only they can be made more efficient by addressing their inherent problems as mentioned previously.

We have previously introduced **the concept of a DAG** data structure, **recursion-DAG** (we used the term "**type-DAG**" [8]), to model recursive algorithms; for example, using a hand-coded implementation of the recursion-DAG, we were able to reduce the integer computation - index computation to access matrix elements - by 30% in matrix multiplication [8]. In this paper, we propose an automatic approach for the determination of the recursion-DAG of general recursive algorithms and we have four main contributions:

1. We present techniques to model the run-time unfolding of recursive algorithms concisely and to abstractly represent function calls with the same division work as a single node in a recursion-DAG.
   The model is a concise and precise representation of the computation and we envision its application as support to drive further compiler techniques or optimizations such as dynamic cache mapping [10], automatic parallelization (e.g., using sophisticated abstract data description [21] and in combination with parallelizing techniques or compilers [33]) and leaf size determination (automatic determination of when the recursion should yield control to blocked algorithms).
2. We present **JuliusC** [12], a (lite) C compiler in which we present an implementation of our techniques for the determination of a **recursion-DAG**. We also show that our implementation, therefore our approach, has practical time complexity and space complexity.
3. We show how the system embodying the above techniques (namely our compiler JuliusC) can be useful as an analysis tool for performance evaluation purposes (profiling) and for software design purposes.
4. To illustrate our techniques and provide a feel for the improvements achievable, we show that we can reduce the division work by 14 or more times for the computation of the binomial coefficients (in fact, the original exponential execution time becomes polynomial) and by 20 millions times for all-pair shortest path of an adjacency matrix (of size 7500×7500). Furthermore, we

```
void fft(MATRIX_TYPE *re, MATRIX_TYPE *im, int n, int stride) {
  int p,q,i,prime,k;

  p = find_balance_factorization(n);
  q= n/p;

  /* leaf or n is prime */
  if (n<=LEAF || p==n ) DFT_1(re,im,n,stride,cos(M_PI/n),sin(M_PI/n));
  else {
    for (i=0;i<q;i++) {  // by column
      k = i*stride;
      fft(re+k, im+k,p, stride*q);
    }
    distribute_twiddles(re,im,n,p,q,stride);
    for (i=0;i<p;i++) {  // by row
      k = i*stride*q;
      fft(re+k, im+k,q, stride);
    }
  }

}
```

Fig. 1. Fast Fourier Transform. The factorization is determined at run-tim and the recursion stops when n is prime or no larger than $LEAF = 5$.

show that the division-work reduction is a function of: the problem size, the algorithm and the pruning technique.

Notice that in parallel applications on large data sets, the division process constitutes the main overhead and it may be driven by a single processor - in a multiprocessor system. A recursion-DAG may be used to reduce the division work, therefore speeding up the sequential execution of the problem division. Also, a recursion-DAG may assist the parallelization process helping the efficient processor allocation to parallel function calls at run-time; for example, it may help reducing the number of process spawns, therefore reducing initialization time and initialization of communications.

We organize the paper as follows. We start in Section 2 with an example. In Section 3, we introduce the related work mostly on the topic of dynamic programming and partial evaluators of λ-calculus. In Section 4, we show the organization of our compiler. In Section 5, we present a set of linear algebra applications where our approach is applied and we derive statistics useful for compile-time and run-time optimizations.

2 An Example

Briefly, we illustrate the application of JuliusC on the recursive Fast Fourier transform in Figure 1. Consider that, at run time, JuliusC reaches a function call *fft(*,*,***200***,1)* (where $*$ is wild, any valid vector pointers). JuliusC then generates a recursion-DAG rooted at the function call *fft(*,*,***200***,1)*. In Figure 2, we present a graphical representation of the recursion-DAG. Each node in the DAG represents a function call (of a certain problem size) during the execution of the function call *fft(*,*,***200***,1)*. Embedded in the recursion-DAG, we may recognize a familiar structure: the **plan**, which is used in scientific libraries - e.g., FFTW [16] - to guide the self-installation of recursive algorithms. The only difference is that the recursion-DAG is not a tree (a plan is a binary tree), because of the node fft<5> associated with the function call fft(*,*,5,*). The node fft<5> is a child shared by two nodes-function calls in the recursion-DAG: fft<10> and fft<20>, which are associated with the function calls fft(*,*,10,*) and fft(*,*,20,*), respectively.

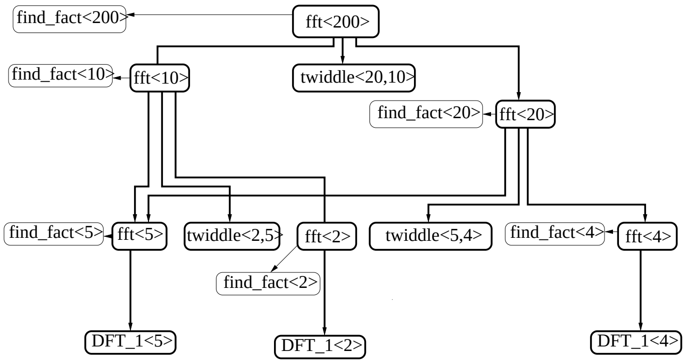

Fig. 2. Recursion-DAG having fft(*,*,**200**,*) as root associated with node fft<200>. To simplify the recursion-DAG presentation, **find_fact** stands for *find_balance_factorization* and **twiddle** stands for *distribute_twiddle* as in Figure 1. Note that every node represents a set of function calls.

In Figure 2, we count only 18 different nodes. Each node is identified by the function name and by an integer summarizing the problem size (i.e., fft<5>). The integer number is determined by the factorization of the problem size of the parent node(s); for example, the number <5> in fft<5> is a factor of both <10> and <20>. So we may store the factorization results in the recursion-DAG, avoiding unnecessary recomputation. For this example, if, in the worst case scenario, the factorization of an integer n takes $O(\sqrt{n})$ operations, the function call fft(*,*,20,*) executes $10*(\sqrt{20}+5\sqrt{2}+4\sqrt{5}) \sim 170$ integer operations for its recursive factorization, and the function call fft(*,*,10,*) executes $20*(\sqrt{10}+2\sqrt{5}+5) \sim 200$ integer operations, for a total of 370 integer operations. Instead, if we compute the factorization of 200, just once, and we store the factors in the recursion-DAG, it takes 14 - i.e., $\sqrt{200}$ - integer operations and 5 store instructions - i.e., 5 factors, respectively.

3 Related Work

Our approach has several similarities with approaches in two very active research areas: one is self-applicable partial evaluation and the other is dynamic programming.

Self-applicable partial evaluation is the problem of optimizing an algorithm when a partial number of arguments is known at compile time and, therefore, specializing a function, or the entire program, to partially precompute the result to the extent feasible at compile time [5, 18, 22]. For example Fibonacci number $F(n)$, [1] if we know that $n = 4$, we may precompute the value and substitute $F(4) = 5$. All authors propose an approach composed of two phases: a static phase and a dynamic phase. The static phase determines what can be computed at compile time and what cannot be computed, called **residual**. Object-oriented class templates, and in particular function specialization in C++, are familiar examples of partial evaluations. Some general purpose compilers also apply partial evaluation for optimizations such as dead code elimination [24].

Our approach is also a combination of static and dynamic phases (analyses). The static analysis takes the input program and annotates the formals parameters of function definitions such as to clearly mark the formals involved in the recursive division process only (e.g., in the function defined in Figure 1, we mark the third formal *int n*). The dynamic analysis execute partially the program, and using the annotations, binds function calls to nodes in the recursion DAG (e.g., the function call *fft(*,*,200,1)* is associated to the node fft<200>, Figure 2). When no arguments is known at compile time, previous techniques are by definition not applicable, however, our technique is applicable and enables us to precompute (at run time) data that will be used during the recursive algorithm execution; in practice, we may specialize the computation of the recursive algorithm at run time.

In **dynamic programming**, there are basically two *philosophies*: function caching and incrementalization. **Function caching** is a top-down approach to solve a problem remembering the result of a function invocation and reusing this result when the function is invoked again with same arguments. For example, Fibonacci numbers $F(4)$ involves the computation of $F(2)+F(1)+F(2)$, the first invocation of $F(2)$ is computed once, and the result is reused for the second invocation [32, 31, 20, 3]. Unfortunately, function caching may require an interpretive overhead to manage and store partial results. In turn, **incrementalization** is a bottom-up approach proposed to reduce or annihilate the overhead typical in function caching [29] reducing the working set of cached function results to a minimum - at any time - and having a fast access to them.

In general, dynamic-programming approaches are not applicable for matrix algorithms, because the final result is highly dependent on the contents of the input matrices (making the *classic* value reuse impossible). However, matrix algorithms have such a regularity in their computation that some of dynamic-programming techniques may be applied, at least in principle. In fact, our technique borrows the same basic ideas of function caching: the recursion-DAG is the collection of all cached function-call results organized in a DAG (even though, we do not necessarely store any function return value). Because we need to store all cached results, incrementalization has no better performance and space advantages than function caching has but at expense of a more complex design and implementation.

[1] $F(n) = F(n-1) + F(n-2)$ with $F(1) = 1$ and $F(2) = 2$

In addition, our approach is not meant for the conversion of loop nests to recursion [28] or vice versa [38], but it offers a means to speed up the conversion process and a quantitative measure of the overhead due to recursion. A compiler may then be able to make an informed choice between recursion or blocked instantiation. Our approach is not meant to optimize the function call mechanism (e.g., register minimization, solving allocation of the actuals-formals, inlining or recursion elimination, state elimination) because modern compiler/hardware technology already exists for these tasks and it is orthogonal to (and it could be use in combination with) our approach.

4 JuliusC: an Overview

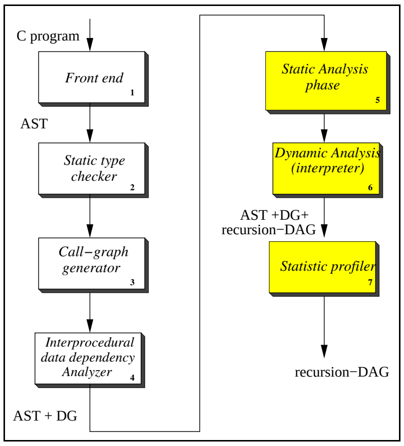

Fig. 3. JuliusC block Diagram. Acronyms: intermediate representation = IR, abstract syntax tree = AST, data dependency graph = DG.

JuliusC is a (lite) C compiler written in C/C++. To simplify the compiler design, we accept a subset of the C language (e.g., no `struct` and no `union` are handled). For linear-algebra applications, our C-language simplifications have little effect on the design and implementation of D&C algorithms (and were chosen to allow a quick demonstration of our ideas). In the longer run, we will port our techniques into more advanced and robust compilers [13] thus avoiding these limitations.

In different phases of the compilation, we manipulate an intermediate representation (**IR**) of the input program. After lexical and syntactical analysis, we create an abstract syntax tree (**AST**). We perform static type checking and we annotate the result of the analysis on the AST. Using the AST for each function definition, we determine all function-call expressions. We then determine a call-graph and its all-pair shortest-path matrix closure. Using this analysis, we may mark whether a function definition is explicitly self recursive, recursive, non-recursive or a **leaf** (a leaf is a non-recursive function that has calls to non-recursive functions only). For example, if the closure matrix shows a loop for a function definition, the function is recursive; otherwise it is not recursive.

We employ a simplified inter-procedural data dependency analysis: the data dependency of each function definition is executed in one pass; arrays are considered as monolithic units (we do not need more accurate analysis because we rediscover the array decomposition from the recursive algorithm, which we assume correct); for loops, we perform the data dependency in one pass (i.e., we do not determine the loop carried dependencies) of the loop body. We would benefit from a more powerful data dependency analysis, which we will plan to obtain by using a more sophisticated infrastructure, but this is not necessary to demonstrate the proposed techniques on useful applications. As a result of the data-dependency analysis, we enrich our IR building a data dependency graph structure (**DG**) upon the AST. In the following, there is our original contribution.

4.1 Static Analysis: the Art of Divide et Impera

Our static analysis summarizes the division process of a recursive algorithm by annotating the **formal** parameters of function definitions (composing the application) with three types of attributes. The attribute-annotation process is based on two steps. First, the formals of self-recursive function definitions are annotated with D&C attributes, and the formals of leaf function definitions are annotated with MO and LO attributes as defined in the following.

```
int factorial(int N) {
 if (N<=1)
    return 1;
 else
    return N*factorial(N-1);
}
```

Fig. 4. *N* is a D&C formal.

Divide and Conquer (D&C): It is a formal that specifies the size of the problem and how the algorithm divides the problem into smaller problems. In practice, a formal is annotated as D&C, if it is used as operand in the conditions of flow-of-control statements, such as if-then-else and while-do statement, and these statements have an execution path leading to a function call of a recursive function. For example, consider the factorial function in Figure 4, the formal `N` is a D&C formal. This is an **inherited** attribute.

Location Operand (LO): a formal is annotated as LO, if it is used in the index computation of a vector. For example, the formal `m` in Figure 5 is annotated as MO and formal `L` as LO. An LO attribute is a **synthesized** attribute.

```
void set(int *m, int n,
         int L ) {
  int i;
  for (i=0;i<n;i++)
    m[i+L] = i;
}
```

Fig. 5. *L* is a LO formal.

The terms *inherited* and *synthesized* come from the way we annotate these attributes when visiting the call graph in the second phase, corresponding to inherited and synthesized attributes in syntax-directed translation.

Matrix Operand (MO): a formal is annotated as MO, if it is a pointer to a vector. For example, the formal `M` in Figure 6 is annotated as MO. This is a **synthesized** attribute.

```
void update(int *M, int n) {
 int i;
 for (i=0;i<n;i++)
   M[i] = i;
}
```

Fig. 6. M is a MO formal

Second, we determine an in-order left-to-right visit of the call graph (e.g., by a depth-first search) starting from the `main()` function definition. The visit determines a tree: the leaf function definitions correspond to the leaves of the tree. As any syntax-directed translation process (e.g., type checking), we visit the tree and we compute inherited and synthesized attributes using the data dependency among function calls. We explain an example shortly.

Consider the example in Figure 7. We annotate the formals of the self-

```
int M[100];

void main() {
  int size;

  initialize(M);
  get(size);
  if (size>0 && size<100)
    put(f(M,size));
}

int f(int *m, int n) {
  if (n<5) {
    return g(m,n);
  }
  return f(m,n/2)+n;
}
int g(int *p, int q) {
  return p[q+1];
}
```

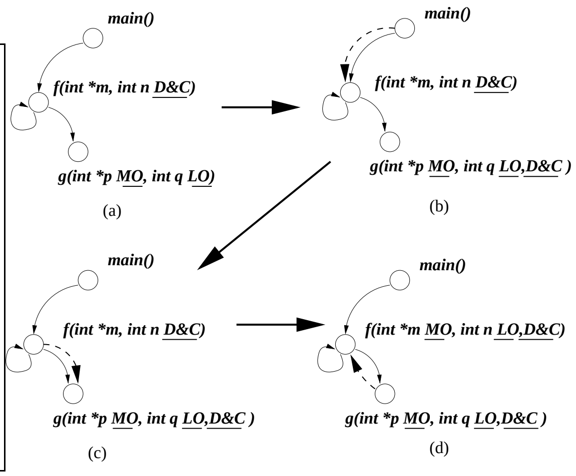

Fig. 7. Example of multiple attributes: n is eventually marked as D&C and LO, the evaluation process is schematically represented in Figure (a), (b), (c) and (d).

recursive function definition *int f(int *m, int n)*: formal `n` is annotated as D&C. We annotate the formals of leaf function definition *int g(int *p, int q)*: the first formal has attribute MO and the second formal has attribute LO (Figure 7 (a)).

We start the second phase of the annotating process from `main()`, which does not have formals. We then visit `f()`: the second formal `q` of `g()` inherits an attribute D&C (Figure 7 (b)) from the formal `n` of function definition `f()`. We then visit `g()` (Figure 7 (c)). We then backtrack to `f()` and we synthesize the attributes from its child `g()`: the first formal has attribute MO, and the second has attribute LO (Figure 7 (d)). Eventually, the formals `n` in `f()` and `p` in `g()` have attributes D&C and LO. For example, `n` in `f()` is an operand in the condition of the only if-then-else statement in `f()` and, indirectly, it is used to access the vector `p` in function `g()`.

An attribute summarizes the use of a formal in the body of the function definition and, in turn, in the functions called as well. In this paper, we present applications for those recursive algorithms having formals with D&C attributes only.

4.2 Dynamic Analysis: Interpretation and Recursion-DAG

During the dynamic analysis, each function call will be associated with a node in a **recursion-DAG** as follows.

When we reach an function call (e.g., `z(i,3,i+3)` in a program), we evaluate the actuals (e.g., `z(10,3,13)`). We look up for the definition of the function call (e.g.,*void z(int i, int j, int k /* D&C */) ...*). Because only D&C formals are used by the recursive algorithm for the division process, we consider the function name and the actual values associated with only D&C formals as the **key** of a function call (e.g., `f(*,*,13)`). Using this key, we look up whether or not a node in the recursive-DAG already exists (with same key). If it does not exist, a new node is added.

The recursion-DAG is an unfolding of the call graph for an initial input, where only the division work is recorded using function caching.

4.3 Static Profiler

The recursion-DAG can be used as a means for a fast and precise collection of statistics about the execution of the recursive algorithm. For example, we may want to compute how many times a function (solving a particular problem size) is called and we just need to exploit the acyclic nature of the recursion-DAG. In fact, we determine a topological sort of the recursion-DAG by a depth-first search. By construction, the function `main()` is associated with the first node and we set its **count** to 1. For every node in the topological sort v, we consider each child u and we update the child count $u.count += v.count$. In this way, while we compute the number of function calls per node visiting each node once, we can compute the total number of function calls and the total number of nodes in the recursion-DAG.

The **reuse ratio**, $\mathcal{R}$, is the ratio between the number of functions calls over the number of nodes in the recursion-DAG. It represents a concise estimate of the work reduction. For example, we may precompute the common computations and store them in the recursion-DAG, the reuse ratio represents how many time we reuse those precomputed values during the execution of the recursive algorithm.

$$\mathcal{R} = \frac{\#\text{function calls}}{\#\text{nodes in recursion-DAG}}. \tag{1}$$

5 Experimental Results

In this section, we show that our approach is practical; that is, the execution time of static and dynamic analysis is negligible. We show that we may exploit reuse of the division work and we present the reuse ratio for 6 recursive algorithms for various inputs.

Our compiler does not yet fully implement the previously developed optimizations (e.g., register allocation) and thus we cannot present here fully automated results of actual speedups. However, our contribution is to present the automatic derivation of the recursion-DAG. The practicality of the overall approach has been already demonstrated previously [8–10].

Static Analysis: The annotation of formal parameters using attributes (e.g., D&C, MO, LO as described in Section 4.1) is based on data-dependency interprocedural analysis and our data-dependency analysis is polynomial in the number of the code instructions.

Dynamic Analysis: At this stage of the project, JuliusC is an interpreter and the recursion-DAG generation is one component of the dynamic analysis (i.e., partial interpretation of the recursive algorithm for some inputs). Therefore, any performance comparisons between the execution time of JuliusC's analysis and the execution time of the recursive algorithm - compiled for the native system - would be inconclusive. For now, we present an asymptotic analysis, which is based only on the recursive algorithm. Ultimately, the recursion-DAG generation time is asymptotically related to the recursion-DAG size. In turn, we may use the reuse ratio to express the relation between recursion-DAG generation - dynamic analysis - and recursive algorithm execution time. Indeed, the execution time of the recursion-DAG generation is $O(1/\mathcal{R})$ of the execution time of the recursive algorithm (Equation 1).

Reuse Ratio: We apply JuliuC to 6 recursive algorithms (for various inputs); three algorithms are from linear algebra; two algorithms are from number theory and one is a classic graph algorithm. The results presented in Table 1 are a significative excerpt and they can be reproduced running JuliusC on-line [12]. Table 1 reports the reuse ratio for recursive algorithms, the ratio represents the possible work reduction we may achieve. We consider briefly each illustrative example as follows.

Binomial is a straightforward recursive algorithm but its time complexity is exponential. Function caching - in JuliusC - allows the reuse of already computed values so as the final time complexity is just polynomial. For the input $\binom{10}{5}$, the reuse is 14; however, larger reuse ratios are achievable for larger problems.

Integer factorization is used in algorithms such as Cooley-Tookey FFT. Given an integer n, we determine the factors p and q so that $n = pq$ and $\min_{p,q} |p-q|$. We determine the factorization for p and q recursively. We analyze an exact and a heuristic factorization (e.g., integer factorization is important in cryptography [26]).

Balanced Cooley-Tookey is the Cooley-Tookey FFT algorithm using balanced factorizations [7, **?**].

Table 1. Reuse ratio $\mathcal{R}$ is a function of the problem size, the algorithm definition and the static analysis.

Notation	Name	$\mathcal{R}$	Inputs
$\binom{n}{k}$	**Binomial**	14 3053	$\binom{10}{5}$ $\binom{20}{10}$
$n = pq$	**Integer factorization**	93 133	$n = 65536$ $n = 524288$
FFT_n	**Balanced Cooley-Tookey**	34 2076 12683	$n = 128$ $n = 5000$ $n = 65536$
$\mathbf{A}^*$	**All-pair shortest path**	60728 23967500	$\mathbf{A} \sim 750 \times 750$, $\mathbf{A} \sim 7500 \times 7500$
$\mathbf{A} = \mathbf{LU}$	**LU-factorization**	16110 93757	$\mathbf{A} \sim 300 \times 300$, $\mathbf{A} \sim 500 \times 500$
$\mathbf{C} += \mathbf{AB}$	**Matrix multiply**	1950 107176 2765800	size of $\mathbf{C}, \mathbf{A}, \mathbf{B}$=$100 \times 100$ size of $\mathbf{C}, \mathbf{A}, \mathbf{B}$=$517 \times 517$ size of $\mathbf{C}, \mathbf{A}, \mathbf{B}$=$1123 \times 1123$

All-pair shortest path is an algorithm based on matrix multiplication and Wharshall-Floyd algorithm [14, 36].

LU-factorization is an algorithm based on matrix multiply with no row pivoting [9, 35].

Matrix multiply is matrix multiplication with matrices stored in non-standard layout [8, 15].

This paper is about how to reduce the algorithm work division, which is intrinsic to D&C algorithms. JuliusC implements an approach to extract and reduce such work division, which is the most difficult part of the implementation.

For completeness, in Figure 8 we show results for a representative example, FFT (the main algorithm is presented in Figure 1, Section 2). In Figure 8, we present an excerpt from JuliusC's output, which is reproducible on-line [12] at the native system - Fujitsu HAL 100MHz. The data dependency analysis takes 0.05 seconds and the annotation of the formals using attributes is negligible (static analysis). The interpretation takes 5.4 seconds (dynamic analysis). The collection of the statistics is negligible. The result of the analysis is a text-based recursion-DAG. Each function call has an entry that we can describe using a single line:

```
name< size > [ number of times this function is called ] |Rec| or |LeaF|
```

The problem `fft()` has size `<6>`; it is called 630 times; the last attribute is `|Rec|`, which stands for *Recursive*. The problem `DFT_1()` has problem size `<1260,3>` (stride and problem size); it is called 3780 times; the last attribute is `|LeaF|`, non recursive.

We use a somewhat limited graphical representation for the final recursion-DAG. We use indentation to present the relation among function calls so we can identify the root immediately: the node `main`. Using this simplified format, two siblings function calls have same indentation and two function calls with a

```
Call Graph from Main ....
----------> get time 0 sec<------
Function calls properties ....
----------> get time 0 sec<------
Data Dependency Analysis ...
----------> get time 0.05 sec<------
Data dependecy result ....
----------> get time 0 sec<------
Marking D&C formals ...
----------> get time 0.01 sec<------
Check the formals on the function definitions ...
Interpretation ...
----------> get time 5.4 sec<------
On count
Reuse Ratio 1007.26
----------> get time 0 sec<------
main<> [1] {} |Rec|
   fft<3780> [1] {0} |Rec|
      find_balance_factorization<3780> [1] {60} |LeaF|
      fft<60> [63] {0} |Rec|
         find_balance_factorization<60> [63] {6} |LeaF|
         fft<6> [630] {0} |Rec|
            find_balance_factorization<6> [630] {3} |LeaF|
            fft<3> [3780] {0} |Rec|
               find_balance_factorization<3> [3780] {3} |LeaF|
               DFT_1<1260,3> [3780] {0} |LeaF|
```

Fig. 8. JuliusC's output excerpt for Balanced Cooley-Tookey

caller-callee relation have different indentation; for example, *fft(*,*,3,*)* is called by *fft(*,*,6,*)*:

6 Conclusion and Future Work

In this work, we have presented an automatic approach to model the computation of recursive D&C algorithms using a DAG data structure, **recursion-DAG**. We show that the approach is practical and it can be incorporated into a generic compiler. We apply our approach to 6 recursive algorithms and we present an estimation of the potential improvements.

In the near future, we will investigate how to use the recursion-DAG as a run-time support for recursive algorithm parallelization and as an alternative means for collecting data currently obtainable by profiling only.

References

1. Lapack – linear algebra package. http://www.netlib.org/lapack/.
2. *Matrix Computations*. Ed. The Johns Hopins University Press, 1996.
3. M. Abadi, B. Lampson, and J.J. Lévy. Analysis and caching of dependencies. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 83–91. ACM Press, 1996.
4. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
5. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of 7th International Conference on Logic for Programming and Automated Reasoning, LPAR'2000*, pages 381–398. Springer LNAI 1955, 2000.
6. J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM Press, 1997.

7. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
8. G. Bilardi P. D'Alberto and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering 2001*, Aarhus, Denmark, 2001.
9. P. D'Alberto. Performance evaluation of data locality exploitation. Technical report. citeseer.nj.nec.com/alberto00performance.html.
10. P. D'Alberto, A.Nicolau, and A. Veidenbaum. A data cache with dynamic mapping. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*. Springer Verlag, Oct 2003.
11. J.J. Dongarra, I.S. Duff, D.C.Soransen, and H.A. van Der Vorst. *Numerical Linear Algebra for Performance Computers*. SIAM, 2000.
12. D'Alberto et al. JuliusC. `http://halps.ics.uci.edu/~paolo/JuliusC`.
13. M. Lam et al., 1994-current. http://suif.stanford.edu/.
14. R.W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6), 1962.
15. J.D. Frens and D.S. Wise. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming*, volume 32, pages 206–216, July 1997.
16. M. Frigo and S.G. Johnson. The fastest fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of technology, Sep 1997.
17. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society, 1999.
18. C. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM Trans. Program. Lang. Syst.*, 14(2):147–172, 1992.
19. F. Gustavson, A. Henriksson, I. Jonsson, P. Ling, and B. Kagström. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In Springer Verlag, editor, *PARA'98 Proceedings. Lecture Notes in Computing Science*, number 1541, pages 195–206, 1998.
20. A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 311–320. ACM Press, 2000.
21. J. Hummel, L.J. Hendren, and A. Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, 1992.
22. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, soft edition, June 1993. http://www.dina.dk/ sestoft/pebook/.
23. I. Jonsson and B. Kagström. Recursive blocked algorithms for solving triangular systems part i: one-sided and coupled sylvester-type matrix equations. *ACM Trans. Math. Softw.*, 28(4):392–415, 2002.
24. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 147–158. ACM Press, 1994.
25. M. Lam, E. Rothberg, and M. Wolfe. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on architectural support for programming languages and operating system*, pages 63–74, Apr 1991.

26. A.K. Lenstra. The development of the number field sieve. volume 1554 of *Lecture Notes in Math.* Springer-Verlag, 1993.
27. B. Kagström P. Ling and C. van Loan. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, Sept 1998.
28. Y.A. Liu and S.D. Stoller. From recursion to iteration: What are the optimizations? In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 73–82, 2000.
29. Y.A. Liu and S.D. Stoller. Dynamic programming via static incrementalization. *Higher Order Symbol. Comput.*, 16(1-2):37–62, 2003.
30. J. Park, M. Penner, and V. Prasanna. Optimizing graph algorithms for improved cache performance. In *In Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.
31. W. Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 269–276. ACM Press, 1988.
32. W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328. ACM Press, 1989.
33. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–83. ACM Press, 1999.
34. Boleslaw K. Szymanski. *Parallel functional languages and compilers.* ACM Press, 1991.
35. S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, Oct 1997.
36. J.D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, volume 19, May 1990.
37. R.C. Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.
38. Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 169–181. ACM Press, 2000.