MatParser: An array dataflow analysis compiler.

Bart Kienhuis
University of California at Berkeley
Department EECS, Cory Hall 524
Berkeley, California, 94720, USA
kienhuis@eecs.berkeley.edu

Technical Report UCB/ERL M00/9

February 29, 2000

Abstract

This paper presents the *MatParser* tool. MatParser is an array analysis compiler that automatically converts an affine nested loop program into a single assignment program. The nested loop programs may contain non-linear operators like div/mod/floor/ceil and stepsizes other than one. The focus of this article is on the software architecture used in MatParser to resolve the data dependencies. Finding that two variables are dependent on each other and at which iteration, is a very computational intensive procedure. MatParser employs a particular linear programming technique to find the data dependencies, based on parametric integer programming (PIP) as proposed by Paul Feautrier. To appreciate the implementation of MatParser, we will explain in this paper in sufficient detail the basics of the linear programming technique used.

1 Introduction

In signal processing, many applications perform large-scale algebraic computations with high data throughput. This can only be sustained by exploiting parallelism and pipelining in the computations. This requires a description that expresses all available parallelism in the computations. Unfortunately, applications are typically written in an imperative language, like C or Matlab and the derivation of a parallel description from these sequential program descriptions is a difficult and tedious task. This paper presents the *MatParser* tool, which automatically converts affine *Nested Loop Programs* (NLPs) into *Single Assignment Programs* (SAPs). SAP expresses all available data parallelism present in terms of the parameters used in the original NLP.

MatParser employs an advanced linear programming technique to find the data dependencies in an NLP as proposed by Feautrier [8]. We have extended this approach such that non-linear operators can be handled like *Div*, *Mod*, *Ceil*, and *Floor*. In addition, For-loops with a stride other than one are allowed. This extension is imple-

mented in MatParser, but will not be discussed in this papers, as it is discussed in detail in [13, 12, 14].

MatParser is the successor of the *HiPars* tool [12]. HiPars is written in *ObjectiveC* [3], which is currently considered an obscure object oriented language. Instead, MatParser is implemented in Java™ [11], making the compiler more accessible and platform independent. MatParser differs from HiPars in that its overall performance is improved and that it has a better-engineered software architecture. The focus of this paper is the software architecture of MatParser, however, to appreciate its implementation, we first describe briefly the technique used in MatParser to find data dependencies.

MatParser has been developed as part of an effort to make the mapping possible of high-performance signal processing (DSP) applications onto new emerging embedded DSP architectures [15, 4]. These new architectures consist of a number of coprocessors, a microprocessor, memory, and programmable interconnect. Since these architectures differ considerably from standard parallel computer architectures, a standard compiler framework is not able to perform the required mapping. Consequently, the focus of MatParser is not to do data analysis on standard languages like *C* or *Fortran* in context of computer architectures as done in standard compiler framework like GCC [21] or SUIF [1]. In general, the dataflow analyses performed in these frameworks are not exhaustive enough for our purpose. We want to know when two variables are dependent, and furthermore at which iteration. The latter requirement is in general not solved by these compiler frameworks.

This paper is organized as follows. In Section 2, the problem of finding data dependencies in NLPs is described as well as the Linear Programming technique used to find these data dependencies. Section 3 describes the basic software architecture of MatParser. Section 4 shows results obtained when running MatParser on a set of NLPs. Section 5 describes conclusions. In the Appendix, the full grammar of MatParser is describes. In addition, a number of examples are given of how to write an NLP for MatParser. Finally, the options MatParser supports are described.

2 Array Dataflow analysis

The MatParser tool belongs to the class of Array Dataflow analysis tools. An *Array Dataflow analysis* is the effort to find the set of all flow dependencies in a program [7]. It finds if two variables are depending on each other, but moreover, at which iteration. In order to find these dependencies, MatParser uses linear algebra techniques. This, however, immediately limits the kind of programs that can be analyzed to the class of affine Nested Loop Programs. Nevertheless, signal/image-processing applications are typically written as affine Nested Loop Programs. To write and develop these applications, the Matlab™programming language [18] is frequently used.

2.1 Nested Loop Programs

Nested-loop programs (NLPs) consist of two kinds of control statements, *control flow* statements and *conditional* statements. The *For-loop* statements describe the control flow and *If/Else* statements describe the conditional statements. Furthermore, NLPs

contain *assign* statements, which take the form of a function-call. Linear expressions used in the various statements need to be affine.

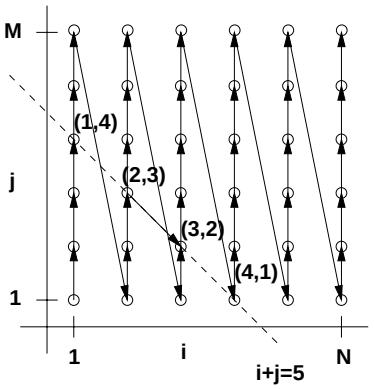
A typical example of a nested loop program is given in Figure 1(a). It shows two For-loops with iterators `i` and `j`. The upper bounds of the For-loops are parameterized in parameter `N` and `M`. This simple example does not use control statements. Inside the two For-loops, the function `Func` is called, which consumes a variable `a` at the Right-hand side (Rhs) and produces a new variable `a` on the Left-hand side (Lhs). The function-call exposes only its input and output arguments, hiding all complexity of how the function is realized.

The statements in the NLP are totally ordered by the sequential semantics of the programming language, which in the case of Figure 1(a) is Matlab. However, a partial execution order exists [2], which is given by the flow dependencies for the variable `a`. A *flow dependency* or *data dependency* means that a variable is first written and later read. If we find all data dependencies in a program, we have obtained the partial execution order that allows for the parallel execution of the program.

The variable `a` describes an array in memory and individual elements are accessed using an *index function* (e.g. $f = i + j$). The variable `a` is used at both the Lhs and the Rhs of the function call. Now consider, for example, variable `a(5)`. The function `func` will use this variable at an iteration (i, j), but also create one at an iteration, say (i', j'). Consequently, a data dependency may exist between subsequent iterations. Finding this data dependency for all iterations and for all variables in an NLP is exactly the problem MatParser solves.

```
for i=1 : 1 : N,
    for j=1 : 1 : M,
        [ a(i+j)  ] = func( a(i+j) );
    end
end
```

(a)



(b)

Figure 1: A Nested Loop Program in Matlab (a) and the iteration space spanned by the two For-loops (b).

To find the solution to the given problem, observe that the For-loops in Figure 1(a) span a 2-dimensional space, the so-called *iteration space* as shown in Figure 1(b). Through the affine index function `i+j`, specific elements of variable `a(5)` are accessed, either

through writing or through reading, as shown by the line $i + j = 5$. Now the question is which iteration instance assigned the value to variable a(5) that is read, for example, at iteration (3,2). From the figure, we see three candidates: (1,4), (2,3), and (4,1). To resolve this choice, we superimpose the order in which the For-loops of the Lhs variable move through the iterations space, as shown by the arrows. This reveals it was iteration (2,3) that assigned a value to `a` at the Lhs that is subsequently read at the Rhs at iteration (3,2). Thus the data dependency matrix $M(i, j)$ between iteration (i', j') and (i, j) becomes

$$\text{M(i,j)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad (1)$$

2.2 Parametric Integer Programming

To find all data dependency relations in an NLP, MatParser employs an sophisticated Linear Programming technique as proposed by Feautrier [5]. In the following paragraph, we sketch the technique used for the example given in Figure 1(a). However, for an in-depth discussion we refer to [5, 6].

Both the Rhs and Lhs variables in Figures 1(a) are enclosed by For-loops that span an iteration space. This space is described mathematically as a polytope $e(I, P)$, where I represents the iterators (e.g. `i` and `j`) and P represents the parameters (e.g. `N` and `M`). Two variables can have a dependency if and only if both access the same memory location. That is, they have the same name and access the same memory location as given by their index functions f and g for respectively the Lhs (e.g. $f = i' + j'$) and the Rhs ($g = i + j$) variables. As shown in Figure 1(b), more than one iteration might satisfy this equality. Therefore, the order imposed by the For-loops is added to resolve which iteration was the latest one that wrote to the memory location. This order is described mathematically by means of the *lexicographical order* [2], denoted as $\ll$.

Using the iterations space I and J of respectively the Rhs and Lhs variable, the index functions and the lexicographical order, we define a new polytope Q as;

$$Q(I) = \{f(J) = g(I) \wedge J_{\ll} \wedge e(J, P) \geq 0\}, \quad (2)$$

in which iteration I needs to be a valid member in context of the polytope $e(I, P)$ of the Rhs variable. Because polytope Q is now lexicographically ordered, finding the "latest" iteration that wrote to a memory location, means finding the *lexicographical maximum*, denoted as;

$$S(I) = max_{\ll} Q(I), \quad (3)$$

in which the solution $S(I)$ is called the *Source function*. This is a symbolic expression describing the data dependency between a Lhs and Rhs variable pair in terms of the parameters P used in the NLP. In case a solution doesn't exist, the Source function is considered undefined.

In defining polytope Q, the lexicographical ordering is used and for an iterator vector I that contains n iterators, there may exist as many as $n + 1$ expansions. Each expansion leads to a different definition of Q and thus solution. To obtain one final data dependence function, the different solutions, i.e., $S_0 \ldots S_{n+1}$, need to be combined into

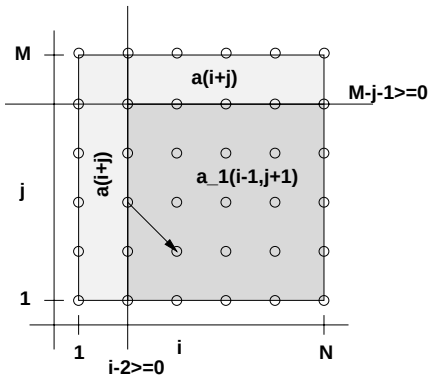
one solution. This combination process can be written as a recursive procedure [5] as

$$M_i(I) = Max_{\ll}(S_i(I), M_{i-1}(I)), \quad i = 1, .., n+1. \tag{4}$$

Besides finding different source functions from the lexicographical expansion, a Rhs variable may have a relation with m Lhs variables. Therefore, the data dependence function $M()$ is potentially the result of combining $m * (n+1)$ source functions.

```
for i=1 : 1 : N,
    for j=1 : 1 : M,
    if i-2>=0,
       if M-j-1>=0,
          [ in0 ] = a_1(i-1,j+1);
       else %% if -M+j>=0;
          [ in0 ] = a(i+j);
       end
    else %% if -i+1>=0;
       [ in0 ] = a(i+j);
    end
    [ out0  ] = func( in0 );
    [a_1( i ,j )] = out0;
    end
end
```

(a)



(b)

Figure 2: A Single Assignment Program (a) and the way the If/Else statements partition the original iteration space of Rhs variable `a(i+j)` into different parts with different data dependence relations (b).

2.3 Single Assignment Statement

A *Single Assignment Program* (SAP) is a program where every variable is assigned only one value during execution of the algorithm [22]. It is equivalent to the Dependence Graph of the original algorithm, which expresses all available parallelism in the algorithm.

In Figure 2(a), the SAP is shown for the NLP given in Figure 1(a). In the SAP, the For-loops and Function call haven't changed. However, the Lhs and Rhs variable did. The Lhs variable `a(i+j)` is replaced with variable `a_1(i,j)`, to assure that variable `a_1` is written only once. The Rhs variable is replaced by an If/Else structure, as defined by the data dependence function $M()$.

The function $M()$ divides the original iteration space into different parts, as shown in Figure 2(b). For each part, a specific data dependency exists, i.e. variable `a_1(i-1, j+1)` describes the data dependence as given in equation 1. Thus, for all points within the dark gray area, the same data dependence applies. Notice that point (3,2) is part of this dark-gray area, as we had already found in Section 2.1. The function $M()$ contains

undefined references because some source functions were undefined. These references are replaced by definition with the original Rhs variable name, in this case `a(i+j)`.

3 MatParser

In the remainder of this paper, we focus on the way MatParser is implemented. The tool is completely written in Java [11], a powerful object-oriented language. To explain the software architecture of MatParser, we use static Unified Modeling Language (UML) diagrams [9]. The MatParser tool consists of three different parts:

1. the **front-end** converts a Matlab program into an internal data structure.
2. the **solver** finds all data dependencies relations between Rhs/Lhs variables.
3. the **back-end** writes out the found SAP in a particular format.

We now look at these three parts in more detail.

3.1 The Font-end, parsing NLPs in Matlab

The front-end of MatParser converts a Matlab program into an internal data structure. The UML diagram for the front-end is given in Figure 3. For the parsing of Matlab, the *JavaCC* parser [17] is used, for which we defined a grammar that is a subset of regular Matlab, enough to describe NLPs.

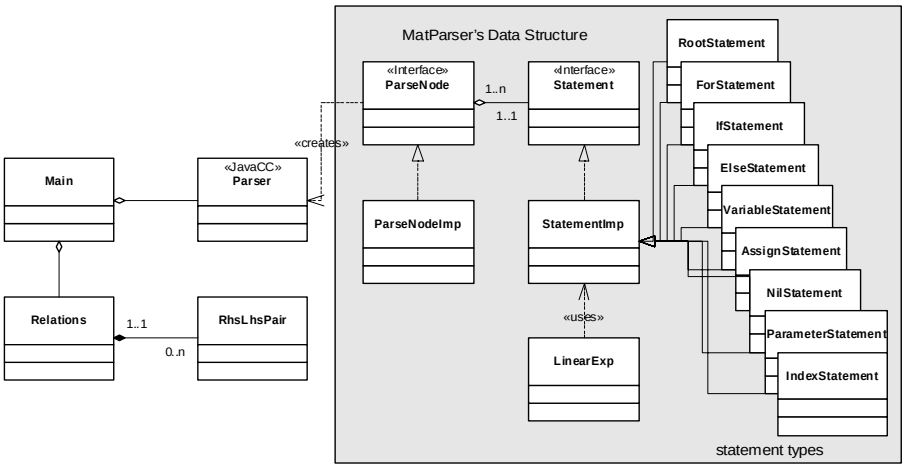


Figure 3: The front-end architecture of MatParser that consists of a parser that reads a nested loop program in Matlab and generates the basic parse tree datastructure.

3.1.1 Data Structure

The parser builds up an internal data structure, which is a parse tree, using `ParseNode` objects. Each `ParseNode` contains an instance of a `Statement`. From class `Statement`, we have derived subclasses to represent the different statement types that may occur in an NLP. For example, class `ForStatement` defines a For-statement and class `If Statement` and class `ElseStatement` represent respectively an If and Else statement. A statement is defined in terms of one of more linear expressions. For a For statement, it is the upper and lower bound expression, for an If or Else statement, it is the condition expression of the statement.

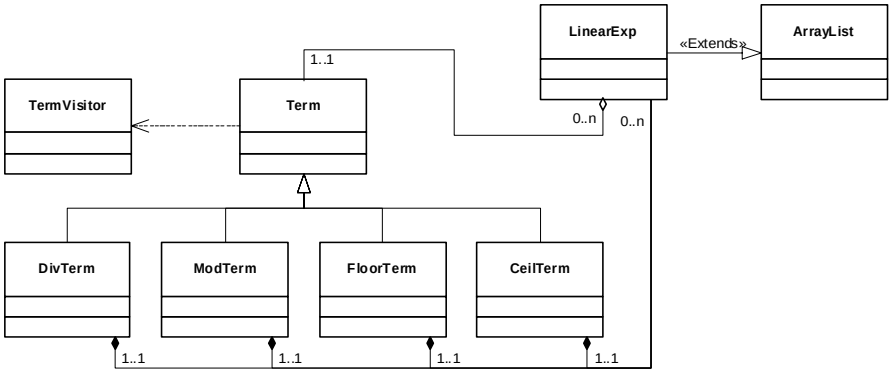


Figure 4: The representation of the linear expressions.

3.1.2 Linear Expressions

A linear expression is represented as an array of *Terms*. Each term represents a signed fractional number or variable. However, we have extended MatParser in such a way, that it can handle the non-linear operators *Div*, *Mod*, *Ceil*, and *Floor*. To accommodate these operators, we have extended the definition of an affine expression. To support these non-linear operators, we derived from Term special terms. Each such term represents one of the four non-linear operators. Each non-linear operator again consists of a linear expression, which is represented by the `LinearExp`. This scheme allows for arbitrary nesting of non-linear operators.

3.1.3 Extensability

The front-end of MatParser can easily accommodate another language other than Matlab. To support, for example, NLPs written in the C language, the grammar of the parser needs to change; however, the tree data structure remains the same.

Once the data structure is set-up, MatParser searches for all possible Lhs/Rhs variable relations in the parse tree that might possibly have data dependencies. For each

valid pair found, i.e., the variables have the same name, a `LhsRhsPair` object is created and stored in the `Relations` object.

3.2 The solver: solving the data dependency relation

The main part of MatParser is the calculation of the data dependency relation between a Lhs/Rhs variable pair. As described in Section 2.2, this means first of all that MatParser needs to find all the Source functions $S_0 \ldots S_{n+1}$ and secondly that it needs to combine them into one data dependency function $M()$. The UML diagram for this part is given in Figure 5.

Object `SolveDataDependencies` controls the two steps: the calculation of a source function and the combining of the source functions into a single source function. The objects involved in finding a source function a colored light gray and the objects involved in combining source functions into a single source function are colored dark gray.

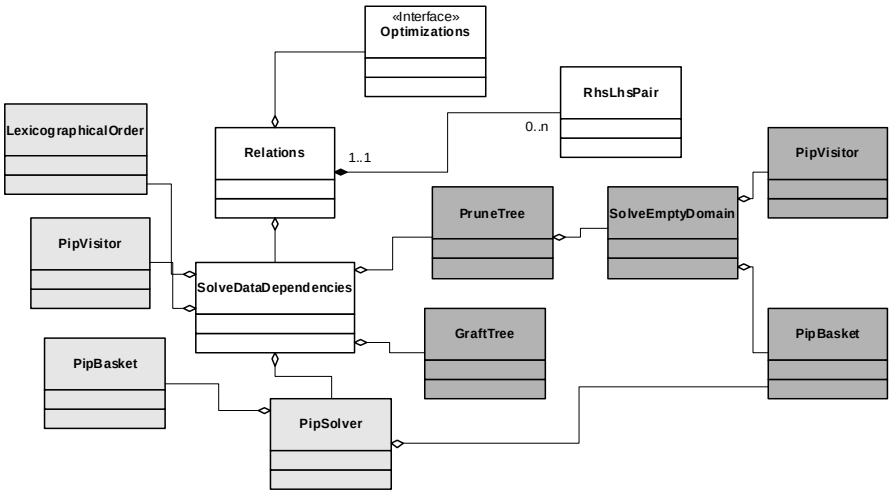


Figure 5: The core part of MatParser that calculates individual source functions and mergers these source function in one data dependency function $M()$.

3.2.1 Find a Source Function

The `SolveDataDependencies` object controls the calculation of the data dependencies. It builds up the polytope Q (See Equation 2) in terms of a list of inequalities, using object `PipBasket`. This object defines a Parameteric Integer Programming (PIP) that needs to be solved. This `PipBasket` object is filled with the correct inequalities using object `PipVisitor`, which is an instance of the so-called *Visitor* design pattern [10]. The visitor design pattern encapsulates in one object, i.e, the visitor, how

a specific operation should behave for the various types presented in a data structure, making the definition and maintenance of type specific operations easy.

For each statement type (See Figure 3), the `PipVisitor` describes how the linear expressions constituting a statement should contribute as linear inequality to the `PipBasket`. The `PipVisitor` uses additionally a `TermVisitor` to describe how the non-linear operators should contribute as linear inequalities to the `PipBasket`. As the `PipVisitor` moves from a leaf node describing a Variable statement to the root of the parse tree, it defines the polytope Q, except for the Lexicographical expansion. This expansion is done separately by object `LexicographicalOrder`. The expansion leads to potentially $n + 1$ different definitions of `PipBasket`.

3.2.2 Solving a PIP system

Each `PipBasket`, that now describes a PIP system in terms of linear inequalities, is sent to object `PipSolver` to find the Lexicographical maximum (See Equation 3). Object `PipSolver` consists of a `Solver` part and a `Decoder` part, which are both defined as interfaces as shown in Figure 6. This makes it easy to use other solvers/decoders when needed. Currently, the original PIP program of Feautrier [8] is implemented as the solver. Alternatively, the *Omega* solver developed by William Pugh [19] could have been implemented. The decoder is implemented as a *JavaCC* parser. It converts the result created by PIP, into a small parse tree describing the source function $S(I)$. The original PIP solver is written in C and to make this program accessible from Java, without much additional coding, the Java Native Interface (JNI) [16] is used.

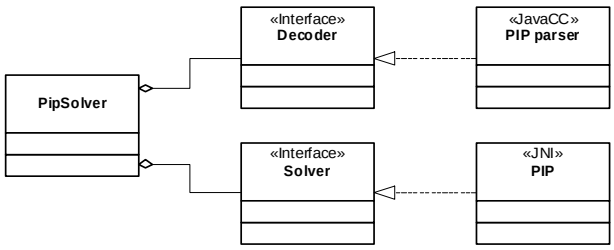


Figure 6: How a PIP system is solved using a particular PIP solver and decoder to convert the found solution back into a parse tree.

3.2.3 Grafting and Pruning Source Functions

The different source functions found as a result of the lexicographical expansion, are combined by the `GraftTree` object (See Equation 4). This object grafts different solution functions into one parse tree describing the data dependence function $M()$. This grafting process is done in a straightforward way, leading very easy to huge parse trees, consuming a lot of memory. In this new derived parse tree, there are many redundant branches. To keep the parse tree small, the `PruneTree` object is used, which prunes

these redundant branches from the parse trees. The pruning of the parse tree is done directly after the grafting process.

A branch is considered redundant when the polytope defined by the leaf node of the branch doesn't contain at least a single point. To determine if a polytope is not empty, the `SolveEmptyDomain` object constructs for each leaf node in the solution parse tree a `PipBasket` describing the polytope $e(I, P)$. If a solution is found by PIP, it means that e contains at least a single point. Otherwise, an empty domain is found that is removed from the solution parse tree.

3.3 The Back-end: generating the SAP

The back-end of MatParser converts an NLP and all its data dependencies found into a SAP description. For this purpose, MatParser uses again a visitor approach. The UML diagram for this part is given in Figure 7. The `MatlabVisitor` generates the SAP. This visitor describes for each statement type how it should be rendered in Matlab. In case the SAP should be written in C, a `CVisitor` needs to be written.

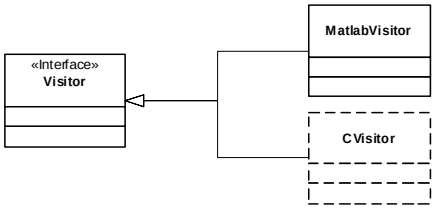


Figure 7: The back-end architecture of MatParser that generates the Single Assignment Code in various formats like Matlab or C.

3.4 Optimizations

In the process of deriving data dependence function $M()$, parse trees are either grafted or pruned in a straightforward manner, allowing for ample opportunity for further optimizations. Hence, before the SAP is written, MatParser first applies a number of optimizations on the function $M()$. The optimizations currently supported by MatParser are given in Figure 8. The optimizations are the removal of redundant If/Else statements (`SimplifySolution`), the removal of redundant Index statements (`RedundantIndexRemoval`), the removal of redundant partitions (`RedundantIfElseStatements`), and the removal of redundant sub graphs (`RedundantSubGraphRemoval`). All these optimizations are implemented in a modular way, which means that each optimization is self-contained; it does not depend on other code other than the basic parse tree data structure. This makes it very easy to extend or improve the suite of optimizations.

Before the optimizations are executed on the solutions parse tree, MatParser first makes a clone of all the linear expressions in the statements. Until now, the parse tree

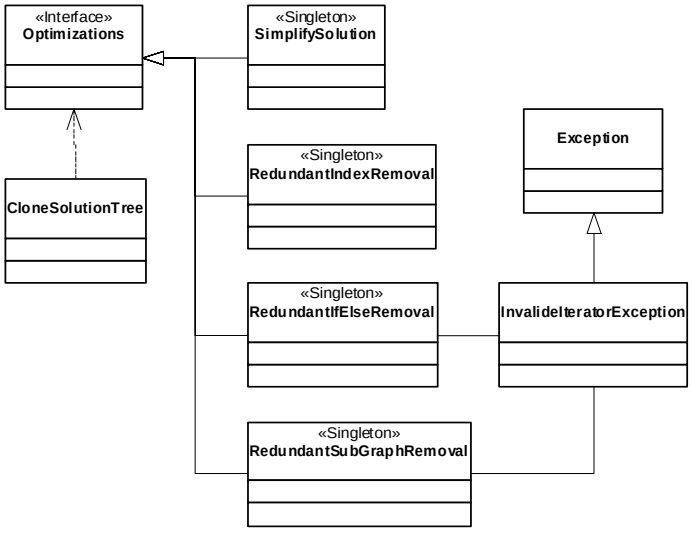


Figure 8: The optimizations implemented in MatParser.

was only extended with partial solutions and statements could be referenced, which save a lot of memory. However in the redundant index statements optimizations, linear expressions used in statements may change as a result of the optimizations. Therefore, a clone is made of each statement for each parse node. This is done by object `CloneSolutionTree]`, in Figure 8.

In some optimizations, such is `RedundantIfElseRemoval` and `RedundantSubGraphRemoval`, the parse tree changes drastically while the optimization traverses the parse tree. In this case, iterators used to walk through the parse tree, become invalid as the parse tree changes. To indicate that the iterators have become invalid, these optimizations throw a `InvalidateIteratorException`. It ensures that the optimizations will start again from the root of the solution parse tree, until a complete solution parse tree is processed by the optimization module.

4 Results

We have run MatParser on a number of NLPs, namely *faddeev*, *gauss* (Gaussian Elimination), *QR* (QR decomposition) and two different versions of *svd* (singular value decomposition). All these NLPs are linear algebra algorithms described in Matlab. While compiling these NLPs into a SAP on a SUN Sparc30, we found the results as shown in table 4.

The first two columns of the table show for each NLP how many Rhs variables are involved that have a dependency with on average how many Lhs variables. The next column shows the average partitioning of the original Rhs variable (See Figure 2(b)).

Algoritm name	Rhs Var.	Lhs Var.	Paritioning of Rhs Var.	Solve Data dependencies.	Solve Empty test	Time per dependency.	Total Time
QR	5	2	3	28	60	226 ms	1 s
gauss	4	3	3	25	36	1875 ms	7 s
faddeev	12	3	4	130	209	6299 ms	1m 15s
SVD (1)	20	6	17	224	5787	13209 ms	4m 24s
SVD (2)	39	20	12	2031	20436	49969 ms	32m 28s

Table 1: Results found when running MatParser on a set of NLPs written in Matlab

The next two columns show how many PIP systems have been solved by the PIP solver to respectively calculate a data dependence or to find an empty domain. Finally, the last two columns show the average time it took to find the data dependence function for a Rhs variable and the total time it took to find all data dependencies in the NLP.

The two SVD algorithms take considerably more time than the other examples, because they use For-loops with a stride and index statements that contain non-linear operators. This introduces lattices on the dense polytopes [13] that make the grafting and pruning of the solution functions considerable more complex and time intensive.

The results in table 4 show that finding all data dependencies is a computationally intensive procedure. In addition, we found that MatParser is very memory intensive, as many parse trees are created and later removed in the graft and prune process.

5 Conclusions

This paper summarizes the PIP-based array dataflow analysis as proposed by Feautrier. Furthermore, it discusses the implementation of the MatParser tool, which implements the PIP-based analysis in Java. MatParser performs the PIP-based dataflow analysis on nested loop programs written in Matlab and convert these programs into single assignment form.

In developing MatParser, we have used extensively modern software techniques like design patterns [10] and UML diagrams [9]. These techniques are used extensively in the design of the Ptolemy II software project, written at UC Berkeley. We were very much inspired by this work [20]. As software techniques, for example, we used interfaces to define contracts between the various components in the architecture to separate their dependence. In addition, we have used design patterns like the visitor, to get easily maintainable and extendable objects, needed when formatting a SAP into different ways, for example, to support different tools that accept SAP code. Finally, we have used UML to discuss and iterate on the basic architecture of MatParser.

We would like to conclude with a number of observations with respect to MatParser. These observations relate to the extensibility of MatParser, its memory use, and the correctness of the obtained SAP descriptions.

5.1 Extensability

A lot of effort has been put in building MatParser in such a way that it can easily be extended with other solvers (for example the Omega solver [19]), or other tools (for example the Polylib package [23] to do empty domain tests). Also, the front-end and back-end are set up such that MatParser can accept NLPs and produce SAP output in another language than Matlab.

5.2 Memory Intensive

The biggest limitation we observed in MatParser is its memory use. MatParser is extremely memory intensive because the solution parse-tree is constantly extended with new partial solutions and pruned for empty domains. The examples given in table 4 use up to 128Mb and the second SVD example didn't even run on 128Mb. We had to run that example on a 512Mb machine.

We have tried to minimize the overall memory use where possible. Nevertheless, we found it very difficult to do memory optimizations. Because of the garbage collector approach in Java, it is often unclear which objects have been cleared, or whether objects are cleared at all. Consequently, memory management is definitely an issue that needs to be further optimized.

5.3 Correctness of the generated SAPs

We are confident that the SAP MatParser is producing is functionally correct. All the SAP programs we obtained from MatParser have been functionally checked with respect to the original NLP program, using Matlab. We found in all checked cases the same input/output behavior.

6 Acknowledgements

We would like to acknowledge the involvement of Ed Deprettere, Edwin Rypkema, Peter Held, John Davis II and Hylke van Dijk. This work was supported in part by the MARCO/DARPA Gigascale Silicon Research Center (http://www.gigascale.org). Their support is gratefully acknowledged.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The suif compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[3] T. S. Corporation. *Objective-C Compiler v4.0*. 1989.

[4] E. F. Deprettere, E. Rijpkema, and B. Kienhuis. Application and architecture modeling for parallel execution of jacobi-type algorithms. In *Proceedings of the 33rd Asilomar conference on signals, systems, and computers*, Pacific Grove, CA, October 20 – 22 1999.

[5] P. Feautrier. Parametric integer programming. *Recherche Opérationelle; Operations Research*, 22(3):243–268, 1988.

[6] P. Feautrier. Dataflow analysis of arrays and scalar references. *Recherche Opérationelle; Operations Research*, 1991.

[7] P. Feautrier. Compiling for massively parallel architectures: A perspective. In *Algorithms and Parallel VLSI Architectures III*, pages 259–270. Kluwer Academic Publishers, 1995.

[8] P. Feautrier. Solving Systems of Affine (In)Equalities: PIP's User's guide. Technical report, Université de Versailles, Saint-Quentin, France, 1996. ftp://ftp.prims.uvsq.fr/pub/logiciels/pip-D.1.tar.Z.

[9] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, 2 edition, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusbale Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Addison-Wesley, 1996.

[12] P. Held. *Functional Design of Data-Flow Networks*. PhD thesis, Dept. EE, Delft University of Technology, May 1996.

[13] P. Held and B. Kienhuis. Div, floor, ceil, mod and step functions in nested loop programs and linearly bounded lattices. In *Algorithms and Parallel VLSI Architectures III*, pages 271–282. Kluwer Academic Publishers, 1995.

[14] B. Kienhuis. Parallelizing nested loop programs containing div, floor, ceil, mod and step functions. Master's thesis, Delft University of Technology, 1994. ET 94-132.

[15] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign*, San Diego, USA, May 3–5 2000.

[16] S. Liang. *The Java Native Interface*. Java Series. Addison-Wesley, 1999.

[17] Metamata Inc. *JavaCC*. http://www.metamata.com/JavaCC.

[18] E. Pärt-Enander and A. Sjöberg. *The MATLAB 5 Handbook*. Addison-Wesley, 1999.

[19] W. Pugh. A practical algorithm. *Communications of the ACM*, 35(8):108 – 114, Aug. 1992.

[20] J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the ptolemy project. Technical report, Gigascale Semiconductor Research Center, University of California, Berkeley, CA, USA 94720, Apr. 1999. Technical Report Series GSRC-TR-1999-01.

[21] R. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA, 1988.

[22] S.Y. Kung. *VLSI Array Processors*. Prentice Hall Information and System Sciences Series, 1988.

[23] D. K. Wilde. A library for doing polyhedral operations. Technical Report Rennes, France, IRISA, 1993. Publication Interne no. 785.

A MatParser Grammar

The subset of the Matlab^{TM}language that MatParser accepts is given here in a BNR formulation. Note that this subset is more restrictive then regular matlab and as a consequence the Matlab MatParser accepts, needs to be rewritten a little.

A.1 Inclusion of standard Matlab code

It is possible to put at arbitrary places standard Matlab code (without any limitation). Ffor example, this makes the testing of NLP and SAP code easy. This Matlab code, however, needs to be placed between two limiters: a `%matlab` and `%end` statement. The Matlab code placed between these two limiters in a NLP will be written out verbatim in the derived SAP.

A.2 MatParser specific statements

There are two statements that are MatParser specific. One is the %parameter statement that defines a parameter and it's range. For example,

%parameter Name min_value max_value;

defines a parameter with a particular name `Name`, that can take any integer value between an including `min_value` and `max_value`. The other statement is the `%function` statement, For example,

%function name;

defines that function `name` is present in the NLP description. The use of this statement is however deprecated. It is included only to be compatiable with NLPs originally written for HiPars. Note that both statements start with %. Their are thus seen by Matlab as comments, when evaluting a NLP.

A.3 Simple and Complex linear Expressions

A difference is made between simple and complex linear expressions. A Complex linear expression may contain non-linear operators like Div/Mod/Floor/Ceil. A Simple linear expression, on the other hand, may not contain non-linear operators.

A.4 MatParser's Grammar

NLP :: [declarations] listOfStatements

declarations :: declaration (declaration)*

declaration :: parameterStatement
| functionDefinitionStatement

matlabStatement :: "%matlab"

```
                                    MATLAB_COMMENT
                          "%end"

parameterStatement :: "%parameter" Identifier fraction fraction ";"

functionDefinitionStatement :: "%function" Identifier ";"

listOfStatements :: statement ( statement )*

statement :: controlStatement
           | functionStatement
           | matlabStatement

controlStatement :: ifElseStatement
                  | forStatement

functionStatement :: indexStatement
                   |assignStatement

forStatement :: "for" Identifier "=" complexExpression ":" Integer ":" complexExpression ","
                      [ listOfStatements ]
                "end"

ifElseStatement :: ifStatement
                                  [ elseStatement ]
                   "end"

ifStatement :: "if" condition ","
                   listOfStatements
elseStatement :: "else" listOfStatements

indexStatement :: Identifier "=" complexExpression ";"

assignStatement :: leftVariableList "=" Identifier rightVariableList ";"

leftVariableList :: "[" [ variable ( "," variable )* ] "]"

rightVariableList :: "(" [ variable ( "," variable)* ] ")"

variable :: Identifier [ "(" [ simpleExpression ( "," simpleExpression)* ] ")" ]

condition :: complexExpression relational_operator complexExpression

relational_operator :: "=="
                     | "! ="
                     | "<"
                     | "<="
                     | ">"
                     | ">="
```

```
simpleExpression :: [ sign ] term ( inearOperator term )*

complexExpression :: [ sign ] termOrOperator ( linearOperator termOrOperator )*

sign :: "+"
      | "-"

linearOperator :: "+"
                | "-"

termOrOperator :: specialOperator
                | term

specialOperator :: "div" "(" simpleExpression "," Integer ")"
                 | "mod" "(" simpleExpression "," Integer ")"
                 | "floor" "(" simpleExpression ")"
                 | "ceil" "(" simpleExpression ")"

term :: fraction [ "*" Identifier ]
      | Identifier [ "*" fraction ]

fraction :: Integer [ "/" Integer ]

Identifier :: IDENTIFIER

Integer :: INTEGER_LITERAL
```

B Nested Loop Program Examples

To get a feeling for what a NLP programs look like that MatParser accepts, we have included three examples of NLPs. The first example describes the Gauss algorithm, the second example describes the QRvr algorithm and the third example describes the SVD algorithm. All three examples were used to get the results found in table 4.

B.1 Gaussian Elimination (Gauss)

```
%parameter N 1 100;

%% Gauss Algorithm

%matlab
u = zeros(N,N);
%end

for j = 1:1:N,
  for i = 1:1:N,
    [u(j,i)] = _ReadMatrix_Zeros_64x64();
  end
end

for i=1 : 1 : N,
  for j=1 : 1 : i-1,
    for k = i+1 : 1 : N,
      [ u(j,k) ] = funcA( u(j,k), u(i,k) );
    end
  end
  for j= i+1 : 1 : N,
    for k = i+1 : 1 : N,
      [ u(j,k) ] = funcZ( u(j,k), u(i,k) );
    end
  end
end

for j = 1:1:N,
  for i = j:1:N,
    [ Sink(j,i) ] = _WriteMatrix_Rout( u(j,i) );
  end
end

%matlab
disp(u);
%end
```

B.2 QR factorization (QRvr)

```
%parameter N 8 16;
%parameter K 100 1000;
for j = 1:1:N,
  for i = j:1:N,
    [r(j,i)] = _ReadMatrix_Zeros_64x64();
  end
end
```

```
for k = 1:1:K,
  for j = 1:1:N,
    [x(k,j)] = _ReadMatrix_U_1000x16();
  end
end

for k = 1:1:K,
  for j = 1:1:N,
    [r(j,j), x(k,j), t ] = Vectorize( r(j,j), x(k,j) );
    for i = j+1:1:N,
       [r(j,i), x(k,i), t] = Rotate( r(j,i), x(k,i), t );
    end
  end
end

for j = 1:1:N,
  for i = j:1:N,
    [ Sink(j,i) ] = _WriteMatrix_Rout( r(j,i) );
  end
end
```

B.3 Singular Value Decomposition (SVD)

```
%parameter M 1 10;
%parameter N 1 5;
%function Pass;
%function vector;
%function thethaSAC;
%function phiSAC;

%matlab
N = 100;
M = 6;
a = magic( M );
%end

for j = 1:1:N,
  for i = 1:1:N,
    [A(j,i)] = _ReadMatrix_Zeros_64x64();
  end
end

for i = 1:1:N,
    [ phi(i) ] = _ReadMatrix_Zeros_64();
end

for i = 1:1:N,
    [ theta(i) ] = _ReadMatrix_Zeros_64();
end


for stage = 1 : 1 : N,
    for i = 1 : 2 : M-1,
        [phi(i),thetha(i)] = vector(a(i,i), a(i,i+1), a(i+1,i), a(i+1,i+1));
    end
    for i = 1 : 2 : M-1,
        for j = 1 : 1 : M,
```

```
     a(i,j), a(i+1,j) ] = phiSAC(phi(i), a(i,j), a(i+1,j));
        end
    end
    for i = 1 : 2 : M-1,
        for j = 1 : 1 : M,
    [ a(j,i), a(j,i+1) ] = thethaSAC(a(j,i), a(j,i+1), thetha(i));
        end
    end
    for i = 2 : 2 : M-2,
        [phi(i),thetha(i)] = vector(a(i,i), a(i,i+1), a(i+1,i), a(i+1,i+1));
    end
    for i = 2 : 2 : M-2,
        for j = 1 : 1 : M,
            [ a(i,j), a(i+1,j) ] = phiSAC(phi(i), a(i,j), a(i+1,j));
        end
    end
    for i = 2 : 2 : M-2,
        for j = 1 : 1 : M,
            [ a(j,i), a(j,i+1) ] = thethaSAC(a(j,i), a(j,i+1), thetha(i));
        end
    end
end
for x = 1 : 1 : M,
    for y = 1 : 1 : M,
        [Sink(x,y)] = Pass(a(x,y));
    end
end

%matlab
disp( Sink );
%end
```

C MatParser Options

MatParser has the following command line options:

options that take an additional argument.

- `--input (-f)` This option is followed by a filename that describes the NLP that needs to be read.
- `--output (-o)` This option is followed by a filename that describes where results, for example a SAP, need to be written. If no output file is supplied, the result is written to standard out.

boolean options

- `--compile (-c)` Compile the NLP into a SAP.
- `--verbose (-v)` Make MatParser verbose. This causes MatParser to produce output showing the progress made in the various step taken to convert a NLP into a SAP.
- `--optimize (-r)` Apply the set of optimizations on the solution trees describing data-dependencies. The optimizations include removing redundant if/else statements, removing redundant index statements, and removing redundant subgraphs.
- `--panda (-p)` This causes the SAP to be written in a format in which all input and output statements that read or write data are written as functions of type `ipd` or `opd`. This format is needed, when the resulting SAP is to be processed further by the Panda tool.
- `--help (-h)` Gives a help text.
- `--version (-V)` Prints out the version number of MatParser.