UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Portable MATLAB Front-end for Tiled Microprocessors

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Hyojin Sung

Committee in charge:

Professor Michael B. Taylor, Chair
Professor Steven Swanson
Professor Sorin Lerner

2009

DEDICATION

To my parents

TABLE OF CONTENTS

Dedication iv

Table of Contents v

List of Figures vii

List of Tables viii

Acknowledgements ix

Abstract x

Chapter 1. Introduction 1

Chapter 2. Design Goals 8
2.A. Performance 8
2.B. Extensibility 10
2.C. Portability 12

Chapter 3. Overall Compiler Structure 15
3.A. Front-end 15
3.B. Back-end 16

Chapter 4. Compilation Phases 19
4.A. Loading and Parsing Functions 19
4.B. Constructing AST 22
4.C. Function Inlining 22
4.D. Building CFG with regions 25
4.E. SSA and SO Transformation 27
4.F. Type and Dimension Inference 28
4.G. Constant Propagation 31
4.H. IR Generation 32

Chapter 5. Performance Evaluation 35
5.A. Evalution Environment 35
5.A.1. Compiler Infrastructure 35
5.A.2. Benchmarks 37
5.A.3. Target Architecture Simulator 39
5.B. Evaluation Results 40
5.B.1. Function Inlining 40
5.B.2. Constant Propagation 43
5.B.3. Function Inlining and Constant Propagation 45

5.B.4. Summary 46

Chapter 6. Conclusion 49

Appendix A. PCODE and TDF Specification 52

A.A. PCODE Specification 52

A.A.1. Tag Format 52

A.A.2. Operation Format 53

A.B. TDF Specification 55

Bibliography 56

LIST OF FIGURES

Figure 1.1. The Raw Microprocessor 3
Figure 1.2. Phase Ordering 4
Figure 1.3. The Role of MATLAB Front-end in Compilers for Tiled Architectures 7

Figure 2.1. Example of Guard Condition Statements 9
Figure 2.2. An Example MPP Function 12

Figure 3.1. Compilation phases in the front-end 17
Figure 3.2. Compilation phases in the back-end 18

Figure 4.1. Function Loading Algorithms 20
Figure 4.2. cos.mpp 21
Figure 4.3. An Example of Abstract Syntax Tree 23
Figure 4.4. An Example of Abstract Syntax Tree 24
Figure 4.5. Standard CFG and CFG with Regions 26
Figure 4.6. Regular Expression for Regions 27
Figure 4.7. SSA and SO Transformation 28
Figure 4.8. MAGICA Input Streams for drv_bayes and bayes Functions 29
Figure 4.9. MAGICA Output Streams for drv_bayes Function 30
Figure 4.10. Pseudo Code for Recursive Mechanisms for Callee Function Inference 33
Figure 4.11. bayes.pcode 34
Figure 4.12. bayes_f2_i2f2f1.tdf 34

Figure 5.1. Execution time of the baseline compiler and the inlining version . . . 41
Figure 5.2. Code Snippet of crnich 42
Figure 5.3. Type inference success ratio 44
Figure 5.4. Execution Time of the Baseline Compiler and the Constant Propagation Version 45
Figure 5.5. Execution Time of the Inlining Front-end and the Inlining & Constant Propagation Version 46
Figure 5.6. Overall Speedup 48

LIST OF TABLES

Table 5.1. Benchmarks used for Performance Evaluation 38
Table 5.2. Benchmark Complexity (measured by # of instructions and # of library function calls) 39
Table 5.3. Elapsed Cycle Time of the Benchmarks Compiled by the Baseline Version and the Inliner-added Version 40
Table 5.4. Speedup of Three Optimized Versions to the Baseline Version 47

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Michael Taylor, whose insightful suggestions and guidance continuously motivated me to improve on the project, and enabled me to finish the thesis. I also want to thank my colleagues, Donghwan Jeon, Ikkjin Ahn and Jinseok Lee, for all their invaluable help and support. Lastly, I'd like to send my special thanks to my family and friends for their unconditional love and encouragement.

ABSTRACT OF THE THESIS

A Portable MATLAB Front-end for Tiled Microprocessors

by

Hyojin Sung
Master of Science in Computer Science

University of California, San Diego, 2009

Professor Michael B. Taylor, Chair

Recently, microprocessor architects have redirected their attention from improving clock frequency to exploiting large-scale parallelism on multi-core processors as a means of continuing Moore's Law. Tiled multi-core processors are one such class of multi-core processors that facilitate compilers' automatic parallelization by providing low-latency communication over on-chip Scalar Operand Networks (SON).

Though parallelizing compilers for academic tiled architectures such as RAW and Wavescalar have shown their potential, they are not portable solutions because they are closely coupled with the source language (typically C) and their own target architecture. Furthermore, their source languages are not suitable for exploiting the full parallelizing power of tiled architectures.

In order to provide portable compiler infrastructure for the class of tiled architecture, we implemented a compiler infrastructure with separate front-end and back-end components. Because MATLAB is the source language, the compiler will benefit from an inherent abundance of parallelism in source codes and conduct an easier analysis without pointers in both performance and complexity.

The thesis presents a complete compiler front-end specifically optimized for MATLAB. On top of conventional front-end tasks, it performs static type and shape inference by using an inference engine, MAGICA. To overcome the limitations of MAGICA, the front-end provides an extended MATLAB format which enables programmers to define type- and dimension-aware MATLAB libraries in a high-level MATLAB.

Performance evaluation measures the effect of function inlining performed in the front-end; function inlining significantly improved performance by 2.38x on average by allowing the inference engine to produce more exact type and shape information.

Chapter 1

Introduction

Until recently, microprocessor technologies have concentrated on improving clock rate as the means of exploiting improvements in silicon manufacturing (Moore, 2000). This approach, however, has produced serious architectural limits. As pipelines grow deeper from the addition of more complicated control logic, the complexity and power density of microprocessors become very difficult to sustain. Furthermore, scaling monolithic processors with central and shared logic are not feasible considering wire and logic delay no more negligible compared to clock speed. Multi-core processors are one of promising alternatives to overcome these complexity and scalability limits.

Multi-core processors combine two or more independent cores into a single physical package. Primary performance gain of multi-core processors results from executing instructions in parallel on different cores, which causes a reduction in execution time. There are various types of multi-core processors in terms of its parallelizing mechanisms. Some processors depend on run-time hardware mechanisms for instruction and data placement, while others use their compilers to perform essential parts of parallelization automatically in compile-time. The main advantage of relying on compilers for automatic parallelization is that they can perform thorough analysis to assign and schedule execution and incur less run-time overheads.

Tiled multi-core processors are a subclass of the multi-core processors that belong to the latter above. RAW (Taylor et al., 2002, 2004), WaveScalar (Swanson, 2006), and TRIPS (Sankaralingam et al., 2003) are among academia's widely-known tiled architectures. A typical tiled microprocessor is composed of relatively simple and identical functional units, which are arrayed in tiles and on on-chip interconnects, as shown in Figure 1.1. The functional units have their own ALU, FPU, registers, and caches, and even their own PC in some cases. They provide an on-chip interconnection network which allows tiles to communicate with other tiles as well as I/O ports for off-chip devices like RAM. Depending on which type of Scalar Operand Network (SON) (Taylor and Lee, 2005) a tiled architecture is built on, instruction assignment/ordering and operand routing are handled either statically or dynamically. According to the AsTRo taxonomy defined in Taylor and Lee (2005), RAW has static assignment/ordering and static/dynamic routing (mostly static) SON, while WaveScalar and TRIPS use compilers only for static assignments and perform ordering/routing dynamically. In any case, tiled architectures heavily depend on compilers when parallelizing instructions and coordinating low-latency communications.

The major advantage of tiled architectures is scalability. Multi-core processors that share hardware resources, like cache, are hard to scale to a large number of cores (i.e., tens to hundreds) because of synchronization and coherence overheads by a centralized logic. In tiled architectures, each tile has its own cache in which to store memory objects, and if another tile needs data in its cache the first specifically sends data to the requesting tile through a low-latency on-chip network. With a sufficiently large per-tile cache, synchronization overheads with memory can be minimized. Due to the distributed nature of the communication, as many tiles as possible can be added to an existing implementation of tiled architecture to provide higher parallelism without increasing its complexity. In addition, because tiled architecture is composed of physically simple function units and they are all distributed, the processor is free from the problems of power and heat density increasing too high on and around complex logic.

In order for tiled microprocessors to really excel in performance by exploiting parallelism, compilers for them play a much more crucial role than conventional com-

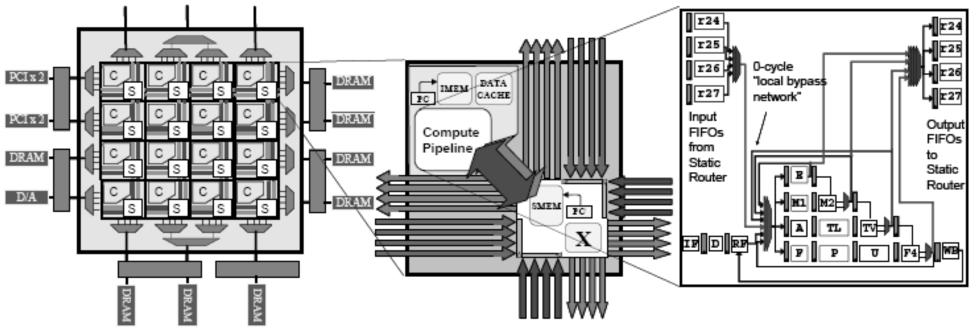


Figure 1.1: The Raw Microprocessor, from (Taylor et al., 2004)

pilers. These compilers are responsible for most of parallelization works, including instruction assignments and operand routing. They typically perform exhaustive code analyses to discover parallelism, and conduct code transformation to make the code even more parallel. Then, they run algorithms to assign instructions and data on tiles to maximize performance while minimizing communications. Figure 1.2 provides an example of these passes implemented in RawCC, a compiler for RAW (Lee et al., 1998). Currently, each tiled architecture has its own parallelizing compiler, and each performs similar parallelization tasks as customized for target architectures. The usefulness of the compilers was once again demonstrated by performance evaluation showing that they can improve performance of applications with high levels of ILP up to $9x$. (Taylor et al., 2004).

Though current compilers successfully provide efficient interfaces for tiled architectures, their source languages pose several reasons to reconsider their applicability. The source languages of current compilers are either C or FORTRAN (Swanson et al., 2003; Smith et al., 2006). These languages are good source languages because they are widely used in many application areas and provide a huge set of benchmarks for the purpose of performance comparison. However, they are not languages originally de-

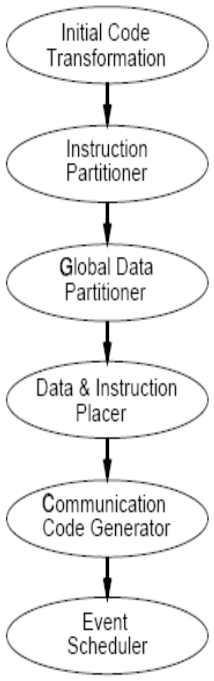

Initial Code Transformation
Instruction Partitioner
Global Data Partitioner
Data & Instruction Placer
Communication Code Generator
Event Scheduler

Figure 1.2: Parallelization Passes in RawCC Compiler, from (Lee et al., 1998)

signed to efficiently handle parallel computation. Parallel or vector operations apply the same operation onto many values in a vector or a matrix. For example, consider a vector operation $A = B + 3$ where A and B are matrices of the same size. This operation adds 3 to every element of B and stores each of them in A at the corresponding index. It is not only much easier to write parallel applications in a language supporting this type of operation, but analyzing and capturing parallelism in the source codes can be performed more efficiently in its compiler. As a result, many application areas, from signal and vision processing to computational biology, which need to execute massive computations in parallel at high speed are written in such language. On the other hand, parallelizable C or FORTRAN applications are very limited. Even though they can express vector operations by repeating scalar operations within nested loops, they require much more complicated data-flow analysis, often over multiple basic blocks, to determine dependency and transform codes to parallel instructions. Therefore, a compiler that takes a language with all the attributes, i.e., which is (1) easy to analyze, (2) has naturally abundant parallelism, and (3) has a large pool of such parallel applications, could exploit the full potential of parallelism available on tiled architectures while maintaining a simple structure.

The thesis proposes to build a portable MATLAB compiler front-end for tiled architecture that will solve the issues of current compilers. MATLAB is a high-level numerical computing environment and programming language (Etter, 1999). It is one of the most widely used and commercially successful languages for performing computationally intensive tasks in diverse areas, such as signal and image processing, computation biology, and financial modeling. There are large growing communities of MATLAB programmers in those areas, and these kind of programs tend to contain a lot of parallelism. Thus, by building a MATLAB compiler, we not only obtain the right material to fully exercise the parallelizing power of tiled architecture, but also connect a large sourcebase of parallelizable code to the architectures that can truly support them. In addition, it is definitely more convenient to analyze and apply optimizations because most of the MATLAB applications are free from pointers. Pointer analysis is usually conservative in identifying independent variables while significantly increasing compiler complexity (Cheng and Hwu, 2000; Burke et al., 1995). If the source language does not involve aliasing, a compiler can perform more aggressive optimizations while keeping itself simple.

Not content to merely provide a functioning MATLAB compiler front-end, we tried to deliver a portable and extensible front-end interface. Using a more portable compiler infrastructure with well-defined standardized abstractions will help reduce the compiler writer's effort as well as expand the accessibility of tiled architecture in general. Extended MATLAB format with syntactic structures defined to force type and dimension to input arguments provide a way to compile MATLAB libraries and generate front-end IRs with complete type information. It helps the front-end to overcome the limitations of the static inference engine and become more extensible. A detailed explanation of each goal is presented in Chapter 2.

Figure 1.3 captures the role of a MATLAB front-end in the portable compiler infrastructure for tiled architectures. It works as a broker, connecting a large sourcebase of parallel applications with the architecture which can execute them faster. Due to its lack of pointers, the front-end performs MATLAB-specific transformations and optimizations while remaining simple and straightforward. Separate front-ends and back-ends which interface through standardized IRs are highly portable across different types of current tiled architectures. In addition, they can expedite design and testing of new tiled architectures.

The remaining part of the thesis is organized as follows: Chapter 2 reviews major design goals of the MATLAB front-end. In Chapter 3, a detailed description of each compilation phase in the front-end is given, along with a big picture of the front-end structure. Chapter 4 presents a performance evaluation of the baseline MATLAB front-end and an optimized version, indicating performance improvement obtained by function inlining. This chapter also includes brief description of the complete MATLAB-RAW compiler framework used for performance evaluation. The conclusion, in Chapter 5, wraps up the discussion and proposes promising future work.

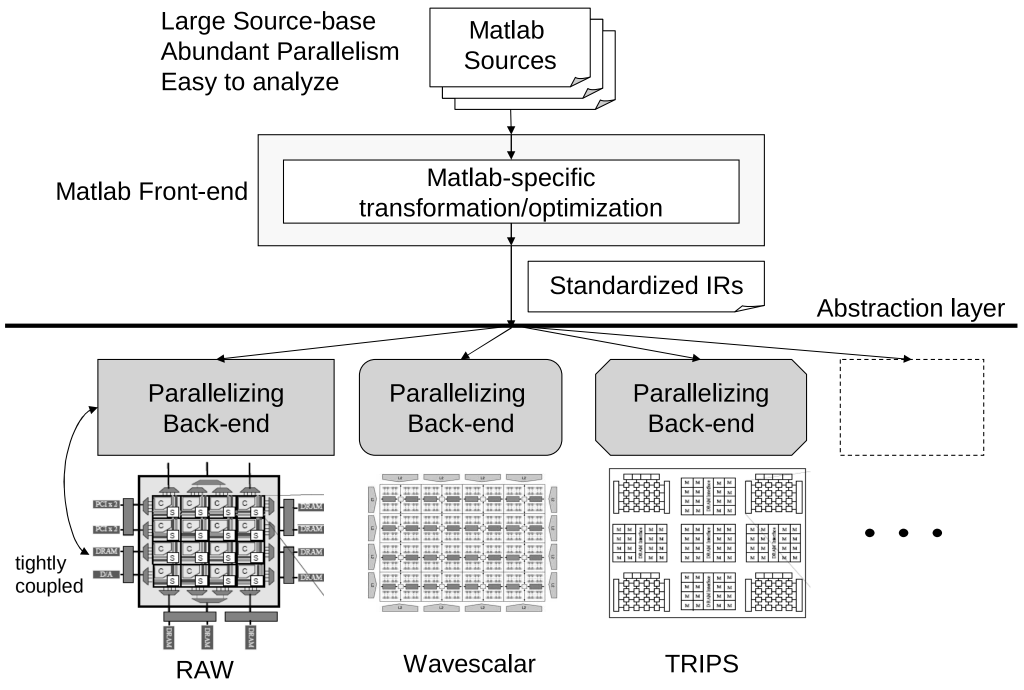


Figure 1.3: The Role of MATLAB Front-end in Compilers for Tiled Architectures

Chapter 2

Design Goals

In this section, three main design goals of the entire compiler structure and, specifically, MATLAB front-end, are reviewed in detail. Each design goal guided our decision process regarding what kind of approach we take to solve imminent problems, what features to include in the compiler, and how to organize them.

2.A Performance

MATLAB is a typeless language which does not require explicit type declaration. Therefore, operators and functions must be overloaded in run-time depending on the type of operands or arguments. For example, a statement $A = C + 3$ depends on the value of C in run-time to execute a simple scalar addition or a multi-dimensional element-wise addition. It also determines the type and dimension of A. As the original MATLAB environment is interpretive, it is not difficult for the MATLAB interpreter to identify the exact type, dimension and value of operands, and execute the matching function in run-time. Compilers which do not perform such run-time analysis, must provide some way to obtain type information. One easy solution is to insert guard instructions before every operation to check types and dimensions of operands and select the right execution function. Figure 2.1 is an example of such implementation. The

```
if (C is scalar)
    A = C + 3;
else if (C is array)
    A = zeros(size(C));
    A(:) = C(:) + 3;
end;
```

Figure 2.1: Example of Guard Condition Statements

clear drawback of this solution is that the run-time overheads which handle all guard branches will significantly degrade performance, and the code size will explode with the added guard structures.

Type and shape inference is one of the most widely-used strategies to improve the performance of typeless languages by resolving type information before the last moment. Static type inference performs the complete inference algorithms statically in compile time (Banerjee et al., 2000; Rose and Padua, 1999; Budd, 1988), while dynamic type inference mechanisms have more simple and straightforward structures, with more information available in run time (Almási and Padua, 2002; Almási, 2001). Static type inference algorithms typically work by running iterative data flow analysis; they begin with certain initial values for variables/expressions and propagate them down the control flow. Ideally, the intrinsic type and shape of all expressions are inferred statically, well before their execution. But, this inference is not always possible, because the type and shape of some variables are determined by which control-flow path is taken in run time. Therefore, for complete type and shape inference, both static and dynamic inference methods should be exploited in conjunction. Statically inferred type and dimension can speed up the execution time by identifying which function to overload well before execution and performing various optimizations, such as pre-allocating memory objects.

In our MATLAB front-end, we perform static type and shape inference using an existing type inference engine called MAGICA (Joisha, 2003; Joisha and Banerjee, 2006, 2002). MAGICA internally operates by modeling the language's shape semantics using algebraic equations expressing the relationships between operands and outputs for each operator. As inference results, MAGICA provides 4 kinds of information: intrinsic type, dimension, size of each dimension, and value range. We assume that all MATLAB source codes should determine explicit type and dimension information after passing the engine. Without this information in compile-time, run-time overheads would be too much to annul performance gain from parallelization. On the other hand, value range and sizes of dimensions are not only less critical but more difficult to infer exactly in compile-time, except certain simple cases. These will have to be resolved by performing dynamic inference in back-ends. It is a design decision made to maximize performance gain by incorporating static type inference engine in the front-end while imposing a minimal restriction on MATLAB sources the front-end can process.

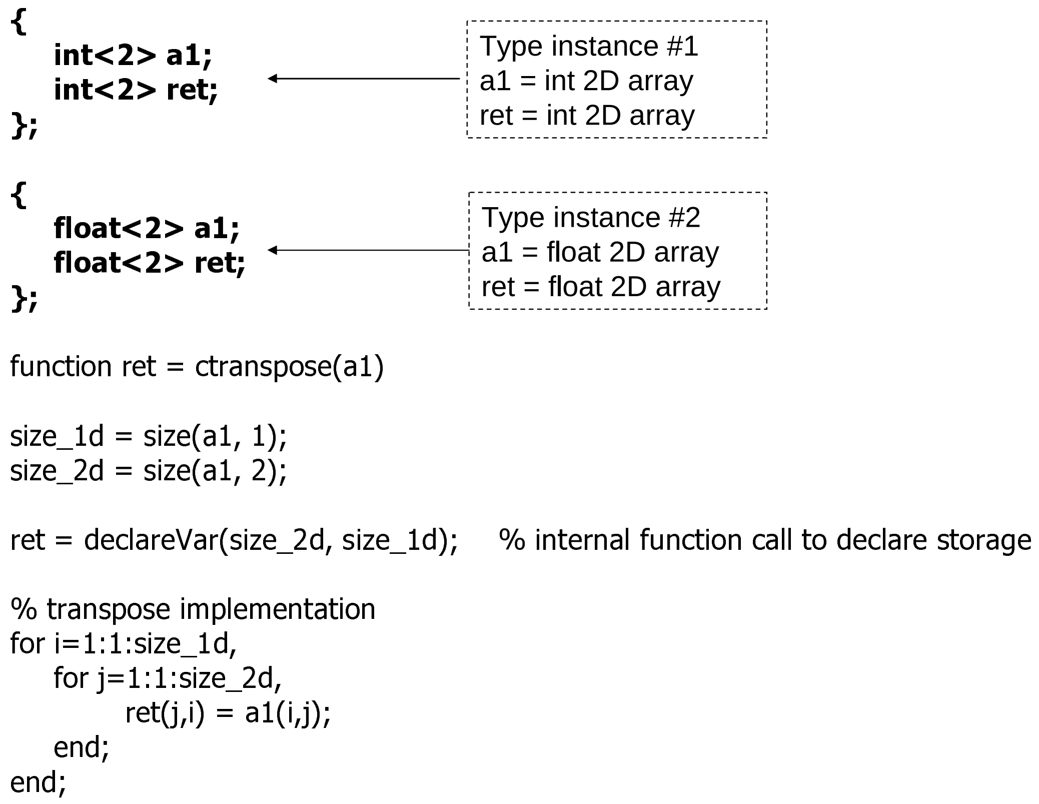
2.B Extensibility

Building an extensible compiler means making it easy to add or improve compiler functionality without disrupting the basic structure. In the MATLAB front-end, we tried to achieve extensibility by supporting MATLAB built-in libraries. MATLAB provides a very large set of built-in libraries, including information about basic mathematical functions and 3D visualization, to help users build their applications more easily. The library is another strong merit of the MATLAB language. However, it is a substantial implementation burden for MATLAB compiler writers, because it has to provide every possible version of one library function, i.e., int/float version and scalar/array version, in order to allow the front-end to link the function in compile-time. Though a scalar is treated as a 1 by 1 array in MATLAB, it is better performance-wise to distinguish between a simple scalar version and a more general array version. Furthermore, as different sets of machine instructions are used to deal with integer and float values, the compiler must generate separate versions to handle each type even if the number of dimensions is the same.

The problem lies in determining which format these library function implementations should be provided in. Each implementation should be aware of the type and dimension of its input arguments, and distinguish itself from other implementations of the same function based on that information. We may write them in PCODE and TDF, which are sufficient to combine the code with type and dimension information. But, this is not a scalable solution, because library writers have to hand-code all TDFs without help from the type inference engine. Also, IR, even if well-defined, is not intuitive to write codes in, and correctness is difficult to prove. Therefore, we propose to provide a convenient way to write MATLAB library implementations, clearly specifying the type and dimension of input arguments and return variables.

The MATLAB++ (MPP) format is defined for this purpose. It has the same MATLAB syntaxes and semantics, except it provides directives to declare type and dimension of input arguments and return variables. The MATLAB front-end takes the directives as type and dimension declaration of those variables and uses this information to initialize them before running the inference engine on the function. Only one PCODE is generated per function since it's typeless, while as many TDFs as the number of type instances are generated per function. Generate PCODE and TDFs are stored in the library directory of the compiler, and linked when a user function calls the library with arguments that have matching type and dimension. If a compiler builder or user wants to add a new library to the list of supported libraries, he/she has only to write its implementations in MATLAB and specify the desired type and dimensions of input arguments and return variables at the head of each file. An example of MPP function is shown in Figure 2.2.

This MPP file generates two ctranspose function implementations, integer 2D array version and float 2D array version. Therefore, there will be one PCODE and two TDFs, one for each version. You can see that type instances are declared before the function body, specifying possible instances of type declarations. declareVar function is a compiler-internal function, valid only in MPP files, which lets back-ends know the size of memory objects to allocate for the variable.



```
{
   int<2> a1;
   int<2> ret;
};

{
   float<2> a1;
   float<2> ret;
};

function ret = ctranspose(a1)

size_1d = size(a1, 1);
size_2d = size(a1, 2);

ret = declareVar(size_2d, size_1d);    % internal function call to declare storage

% transpose implementation
for i=1:1:size_1d,
   for j=1:1:size_2d,
         ret(j,i) = a1(i,j);
   end;
end;
```

Figure 2.2: An Example MPP Function: ctranspose.mpp

2.C Portability

Our goal to build a portable compiler infrastructure is inspired by recognition of the lack of portability in current compilers for tiled architectures. Each of the compilers for existing tiled architectures is very tightly coupled with and optimized for its source language and target tiled architecture. This arrangement may be natural in the sense that these research architectures are developed in different labs, and each group had to concentrate on developing high-performance compilers for their own architecture. But considering the general class of tiled architecture, it is clear that building a more portable compiler infrastructure which can be easily adopted by any type of tiled architecture will expand the accessibility of tiled architectures in general. In addition, its contribution to saving architects' and compiler writers' effort will be significant, considering that building a complete compiler is a very time-consuming job.

High portability of the compiler infrastructure is achieved by completely separating the job of building front-end and back-end through defining general but powerful textual Intermediate Representation (IR) formats. The role of front-end and back-end can be characterized as source-oriented and target-oriented, respectively (Cooper and Torczon, 2003). A front-end typically performs a variety of code analyses and transformations which can extract useful meta-data from the source code or transform it into a more compiler-friendly format. For example, converting the source code into a Control Flow Graph (CFG), to expose control flows, and applying Static Single Assignment (SSA), to simplify def-use chains, are usually performed here. In addition to these, language-specific passes can be added to provide useful, and sometimes necessary, information for further optimization and parallelization in back-ends. Then, back-ends go through various architecture-specific optimizations based on the source codes transformed and annotated by the front-ends, and finally generate sequences of parallel instructions for the compiler's target architecture.

To make each end truly portable, it is vital that we define the proper file format of IR between front-end and back-end. As this format is shared by many languages, it should be general and flexible enough to effortlessly express any language. To avoid limiting types of information allowed in IR, the format should be easy for each front-end to define and add additional language-specific information to, which may then be used by back-ends that are aware of the information. At the same time, the standardized structure should force front-ends to include at least universally crucial information for back-ends. For example, variable type and dimension of variables can be considered necessary computations for all back-ends to perform optimizations. By defining such IR file format, front-end and back-end become separate software components that communicate through off-line files. Therefore, once a front-end is built to produce correct IR output files, it can be paired up with any back-end which can properly process them.

Two types of IR, Polymorphic Code (PCODE) and Type Definition (TDF), are currently defined for our compiler with these requirements in mind. Both IRs are defined in XML format, which is a very flexible and structured way to represent data. PCODE is *polymorphic* it can be shared by multiple TDFs. We'll look into how multiple TDFs are generated for a function in detail later in this chapter.

PCODE basically represents a CFG generated by front-ends. But it also provides high-level semantics such as loops and conditional statements which are common in programming languages as region structures. These high-level structures can help back-ends perform multi-block or global data-flow analyses. Otherwise, back-ends have to reconstruct these high-level structures from basic blocks.

TDF is a map of variables and their type and dimension information. All front-ends are responsible for providing TDFs. In TDF, type and dimension are mandatory for every variable, while the size of each dimension is optional. Our MATLAB front-end does not provide optional information, but other languages such as C and JAVA can easily identify all three information types from variable declaration statements. In that case, back-ends can perform additional optimizations, such as memory prefetching and compaction, with the knowledge of exact sizes of variable dimensions. PCODE and TDF specifications are presented in Appendix A.

Chapter 3

Overall Compiler Structure

The thesis focuses on the front-end of the compiler and how it provides an efficient and portable interface for MATLAB. To evaluate performance of the front-end, however, we need a complete compiler infrastructure – including a back-end that generates machine instructions. Therefore, we paired the MATLAB front-end up with a RAW back-end, which is a parallelizing back-end built for RAW (Taylor et al., 2002) tiled architecture. Note that we are using a new RAW back-end under development at UC San Diego under the guidance of prof. Michael Taylor, instead of the one presented in (Barua et al., 2001). In this chapter, we will give a brief overview on the two components of the compiler.

3.A Front-end

The MATLAB front-end is responsible for translating MATLAB source codes into standardized IRs. The structure of the front-end passes is presented in Figure 3.1.

The front-end takes MATLAB source files with `.m` extension and MATLAB++ (MPP) source files with .mpp extension. Parsers and lexers, automatically generated from rule definitions by Java ANTLR (Parr, 2007) package, generate parsed tree for functions. Abstract Syntax Tree (AST) is constructed from the parsed tree. The baseline version of the compiler does not include any optimization pass to verify that IRs are correctly generated even when type inference is performed with the most conservative initial information. This enables us to show how performance can be improved by function inlining and constant propagation, i.e., by removing function calls and exploiting more precise type inference results as a result. Basic data flow analysis to construct a CFG from the AST is performed, after which Static Single Assignment (SSA) and Single-Operator (SO) code transformations are applied to the CFG. Then, the front-end generates input for the type inference engine, MAGICA, and executes the engine to get inference results. Front-end output includes two types of IR: PCODE and TDF. PCODE is generated from the CFG, while TDF lists type and dimension information obtained from the inference stage. More detailed description of each compilation phases will be given in Chapter 4.

3.B Back-end

The RAW back-end of the compiler performs a variety of optimizations to generate parallelized instructions, closely optimized for the target architecture. One of its most important roles is determining how to perform instruction assignment and scheduling over tiles to minimize communication overhead and maximize parallel execution. The compilation passes in the back-end is given in Figure 3.2.

The back-end takes PCODE and TDF as its input to apply various optimizations and ultimately generate machine instructions. At the first desugaring stage, the compiler combines PCODE and TDF to generate Single-Threaded Code (SCODE) IR, and allocate storage for memory objects. While SCODE IR is single-threaded and therefore targeted for a single tile, Multi-threaded Code (MCODE) IR (output of the next parallelization stage) is multi-threaded code which runs on multiple tiles. A variety of fine-grained parallelism techniques are exploited for more efficient and parallel assignment and scheduling of instructions and data over the tiles. Routing instructions are also generated at this stage. At the final targeting stage, RAW assembly is generated from

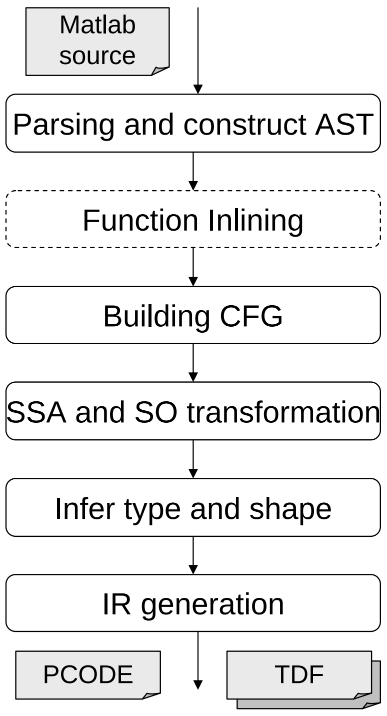


Figure 3.1: Compilation Phases in the Front-end

MCODE by applying peep-hole optimizations and performing register allocations.

Details of design and implementation of the back-end is out of the scope of this thesis.

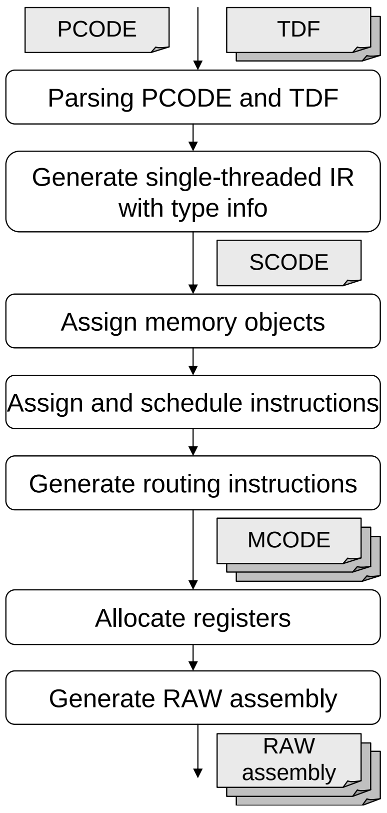


Figure 3.2: Compilation Phases in the Back-end

Chapter 4

Compilation Phases

Front-ends in modern compilers usually include the following compilation phases: parser and lexer to generate an AST; control- and data-flow analysis to generate an CFG and apply code transformations; and an IR generator to transfer information necessary for the back-end to perform further optimizations. On top of the common features, the MATLAB front-end includes a language-specific compilation phase that performs type inference for all the functions in the sources. In this chapter, we will examine each compilation phase in the front-end from input to output.

4.A Loading and Parsing Functions

The front-end begins the compilation process by parsing the MATLAB file, whose name is given to the front-end as a command-line argument. There are two types of source files: M(MATLAB) files with `.m` extension, and MPP(MATLAB++) files with `.mpp` extension. M files are normal MATLAB source files, while MPP files are extended MATLAB source files which generate IRs for MATLAB libraries, as we've seen in Chapter 2. M files can contain more than one function. The function with the same name as the file is the root function, while the others are local functions visible only within the file. The front-end starts parsing from the root function, identifies func-

```
// (1) check if it is a simulator-supported C library function (applies only to MPP)
if (function name starts with "__")
        check native.xml
        if (function name is in the file)
                // it is a native function
                return;
        else
                throw exception;
// (2) check if it is a Magica-supported Matlab built-in library
if (function name is in the built-in list)
        // it is a supported built-in function
        return;
// (3) check if it is one of the sub-functions in the same file
If (function name is in the sub-function list)
        // it is a sub-function
        return;

// (4) check files in the same directory (.m)
        if (filename.m exists)
                load the file;
        else
        // (5) check the library directory
                if (lib_dir/filename.mpp exists)
                        load the file;
                else
                        // the file does not exist
                        throw exception;
```

Figure 4.1: Function Loading Algorithms

tion calls, and recursively loads source files for the functions to parse provided that they are user-defined functions rather than MATLAB library functions. The libraries will be linked automatically by the compiler in later stage. The algorithm used to search and load files is presented in Figure 4.1.

The current RAW back-end is designed to evaluate the performance of the cycle-accurate RAW simulator Barua et al. (2001). The simulator supports a set of C standard library functions by providing RAW assembly implementations of them. We can accelerate execution by linking these implementations directly rather than compiling

```
{
  int a1;
  float ret;
};

{
  float a1;
  float ret;
};

function ret = cos(a1)

ret = __cos(a1);   % calling C library cos function
```

Figure 4.2: cos.mpp

and linking their MATLAB implementations. Therefore, some of the MPP libraries are implemented by invoking the functions. C library functions are distinguished from MATLAB functions with a double underscore prefix, and the list of supported C library functions are kept in the native.xml file in the library directory to make it easy to update the list. On the other hand, there are MATLAB library functions supported by the type inference engine, MAGICA. The return value of these functions can be inferred automatically by MAGICA. In case the callee function is one of the other functions in the same file, it must have been parsed together when the root function was parsed. Therefore, these three types of functions do not require additional file loading. When searching for a file containing functions, the front-end first searches the source directory where the root file is located. If there's no matching file, the front-end then looks at the library directory. The front-end stops file searching and throws an exception when the library directory search fails.

The MPP library function as seen in Figure 4.2 shows how the code invokes C library functions provided by the RAW simulator. The __cos function call is transformed into native instruction in PCODE IR, which is then replaced with the C library function later in the back-end.

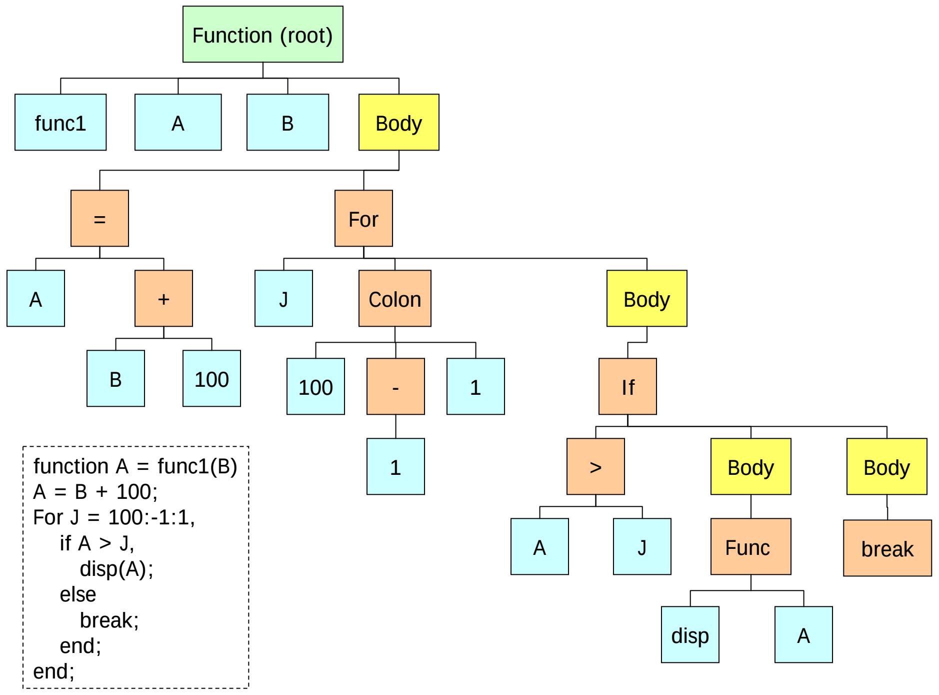
4.B Constructing AST

In this phase, parsed trees generated from the parser are converted into Abstract Syntax Trees (ASTs). An AST is a tree representation of the syntax of source codes. All nodes in the AST are objects encapsulating various syntactic primitives in the source codes from constants, and include variables to control structures like for, if, and while. The nodes are hierarchically structured, starting from the unique root node containing function declaration information at the top, intermediate body nodes containing a block of statements to leaf nodes representing ID and constants. Figure 4.3 shows a sample AST constructed by the front-end. We use a visitor-patterned (Martin, 2003) interface to traverse the AST. The visitor design pattern is good for separating an algorithm from the object structure on which it operates. By implementing the interface and adding the desired action in each visitor function, an optimization and transformation filter can be easily built.

4.C Function Inlining

Function inlining is one of the most common compiler optimization techniques. It replaces a function call with the actual body of the function. Its benefit comes from avoiding bookkeeping overheads involved in context switching at function call-sites, especially when the callee functions are so short that the overheads dominate the overall function execution time. The optimization pass can be performed at various stage of compilation, but the code after inlining can reveal hidden possibilities for optimizations. In the MATLAB front-end, whether to put function inlining before or after type inference can significantly influence the accuracy of type information at the end of the front-end.

As mentioned in Chapter 3, the baseline implementation of the front-end does not include any optimization. The version that lacks function inlining can contain function calls, and the front-end must arrange separate type inferences for callee functions.



```
function A = func1(B)
A = B + 100;
For J = 100:-1:1,
    if A > J,
       disp(A);
    else
       break;
    end;
end;
```

Figure 4.3: An Example of Abstract Syntax Tree

Although MAGICA does not provide type inference results for variables in callee functions, we need the information to generate IRs for the functions. Therefore, we add initialization code for the input arguments of the callee functions using actual arguments at the call-site to start type inference on the functions. A pitfall to simply assigning actual arguments to formal arguments is that the assignment may cause over-specialized inference results.

Let's see the example in Figure 4.4. For certain combinations of argument values, the return variable can be inferred an integer, as seen in the figure. We cannot reuse this type inference result, and thus corresponding TDF safely for every call instance of this function, because the inferred type is not true for other cases where a float value is assigned to the return variable. This method requires the compiler to perform type

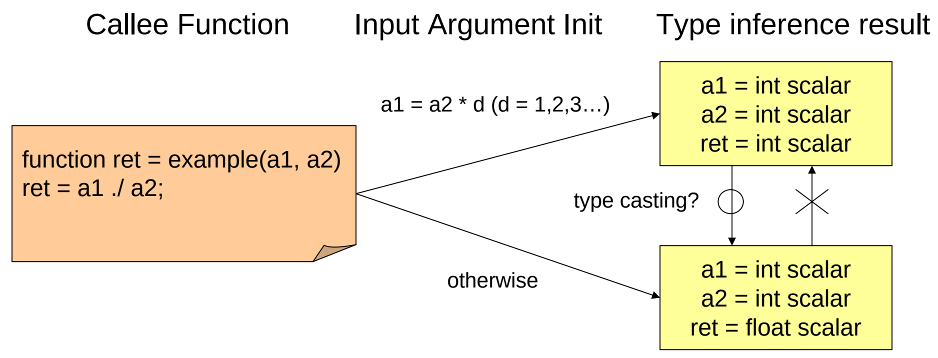


Figure 4.4: An Example of Abstract Syntax Tree

inference of the same function as frequently as it is called. To avoid redundant type inference and improve reusability of IRs, we assign more conservative information to input arguments. The type and dimension are retained from actual arguments, while the value is abstracted to a range. For example, an integer scalar argument is initialized as an integer scalar with value range from -25536 to 25536. This will enable us to avoid over-specialization and get more general type inference result though it may suffer from type casting from float to integer for some cases performance-wise. Therefore, this version sets up the compiler's baseline performance without conducting any optimization related to type inference.

Function inlining is performed at the AST level. The front-end traverses ASTs to find function calls, and it recursively merges the AST of every user defined function called into the AST of the caller function. As a result, one large root function AST is produced without user-defined function calls. This inlined version can give much more accurate type and dimension inference results without losing any value information on function call-sites, in exchange for increased code size. Many previous MATLAB compilers, including FALCON (Rose and Padua, 1999), took this approach to maximize performance gain from static type inference.

4.D Building CFG with regions

In this stage, the compiler front-end constructs a Control Flow Graph (CFG)s from an AST. CFG essentially shows all control paths that can be taken in program execution, and is crucial to many compiler optimizations and static analysis. In our front-end, a basic CFG is extended with the concept of regions.

In converting AST to CFG, syntactic structures such as loop and conditional statements are all abstracted to conditional/unconditional branches at the end of basic blocks. This syntactic information can help back-ends to perform a variety of transformation and optimization operations. While CFG is inherently suitable for finding local optimal solutions within the boundary of each basic block, there are many cases which benefit from global analysis across basic block boundaries. For example, to perform loop fusion to reduce the number of loops and increase efficiency of each individual loop, global knowledge of loops are necessary. Because loops consist of one or more basic blocks, local basic block analysis can not properly capture the relationships between sequential, independent loops. If only a basic CFG is provided to back-ends, they might have to restore the source code's original syntactic structures by running a full data-flow analysis. There was similar approaches to provide both high-level and low-level information of source codes in a single IR (O'Brien et al., 1995). The TOBEY compiler has two types of IR, XIL and YIL, which functionally correspond to Instruction and Region in our PDF respectively. While YIL strictly sticks to retain source-level syntactic structures such as loops and conditionals, our approach is different in that it uses a unified concept of region to express all kinds of syntactic structures, and actually removes all control-flow branches by imposing valid inner-region structures with implicit predecessor and successor region links.

Figure 4.5 compares fragments of two types of CFG (standard CFG and CFG with regions) for bayes function. CFG with regions is built based on standard CFG,

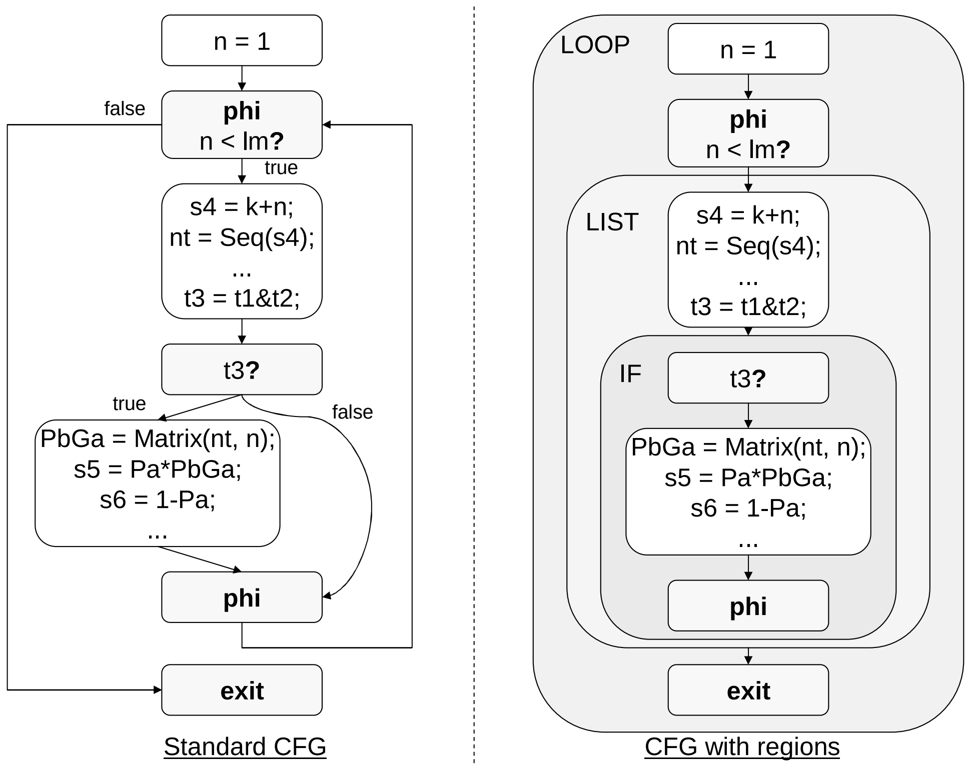


Figure 4.5: Standard CFG and CFG with Regions

wrapping basic blocks with regions. Therefore, it can be traversed either in AST style (statement-by-statement) or CFG (block-by-block) style. There are four types of regions: NODE, LOOP, LIST, and IF. Each region is defined using regular expressions in Figure 4.6. A NODE region represents a basic block without branches, while a LIST region contains a list of the other three types of regions. LOOP regions are supposed to have a preheader NODE region, a condition NODE region, a loop LIST region and a loop exit NODE region. If there is no instruction to put in a region, the region will just be left empty. IF regions have a condition NODE region and a true LIST region, while a false LIST region is optional.

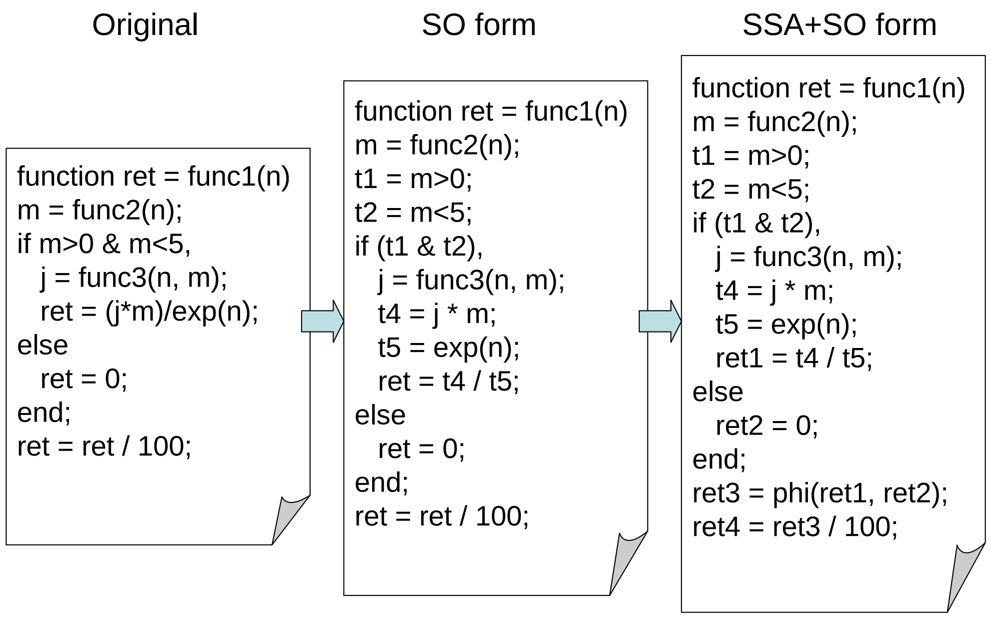
```
REGION ::= LIST_REGION | NODE_REGION | LOOP_REGION | IF_REGION

LIST_REGION ::= (NODE_REGION)*

NODE_REGION ::= (INSTRUCTION)*

LOOP_REGION ::= NODE_REGION          LIST_REGION    NODE_REGION
                preheader/condition  body           exit

IF_REGION ::= NODE_REGION   LIST_REGION   (LIST_REGION)?
              condition     true body     false body
```

Figure 4.6: Regular Expressions for Regions

4.E SSA and SO Transformation

Static Single Assignment (SSA) (Cytron et al., 1991) and Single Operator (SO) transformations are algorithms that transform codes into more normalized forms in order to more easily perform analyses and optimizations, as well as generate instructions (Cooper and Torczon, 2003; Kennedy and Allen, 2002). At the same time, both transformations are required by MAGICA to be applied to its input program (Joisha and Banerjee, 2002).

The SSA form of a function has exactly one assignment for every variable. Every time a variable is redefined, a unique subscript is added to the original name. Therefore, every definition has its own version. In SSA form, def-use chains are very simple and have only one def element. The primary benefit of SSA comes from how it facilitates a variety of data-flow analyses and optimizations based on them, including constant propagation, dead code elimination, strength reduction and etc. SO transformation divides operations with more than one operator into a sequence of single-operator instructions. In this process, temporary variables are generated by the compiler to store intermediate computation results. Since many machine instructions are in SO form with a maximum of two operands, this transformation helps the compiler directly generate machine



```
function ret = func1(n)
m = func2(n);
if m>0 & m<5,
   j = func3(n, m);
   ret = (j*m)/exp(n);
else
   ret = 0;
end;
ret = ret / 100;
```

```
function ret = func1(n)
m = func2(n);
t1 = m>0;
t2 = m<5;
if (t1 & t2),
   j = func3(n, m);
   t4 = j * m;
   t5 = exp(n);
   ret = t4 / t5;
else
   ret = 0;
end;
ret = ret / 100;
```

```
function ret = func1(n)
m = func2(n);
t1 = m>0;
t2 = m<5;
if (t1 & t2),
   j = func3(n, m);
   t4 = j * m;
   t5 = exp(n);
   ret1 = t4 / t5;
else
   ret2 = 0;
end;
ret3 = phi(ret1, ret2);
ret4 = ret3 / 100;
```

Figure 4.7: SSA and SO Transformation

instructions from IR instructions. Figure 4.7 shows how a sample MATLAB code is transformed through SSA and SO transformation stages. Phi functions generated by SSA transformations will be removed by back-ends at the code-generation stage. After this stage, the CFGs for source functions are ready to be given to MAGICA as input and printed as PCODE IR files.

4.F Type and Dimension Inference

As discussed in previous sections, an existing type inference engine called MAGICA is incorporated in the front-end of the MATLAB compiler for static type inference. It was developed as an add-on module for MATLAB (Joisha, 2003; Joisha and Banerjee, 2002) to infer type and shape information from source codes. It currently supports 70 MATLAB built-in libraries, and infers the type and shape of library return values automatically.

```
inputArgs[drv$bayes] ^= {};
outputArgs[drv$bayes] ^= {};
statements[drv$bayes] ^:= Sequence[assignment[$$lhs
:> drv$bayes$r1$ssa0, $$rhs :> rand[1, 30]],
assignment[$$lhs :> drv$bayes$r2$ssa0, $$rhs :>
mtimes[drv$bayes$r1$ssa0, 3]], assignment[$$lhs :>
drv$bayes$r3$ssa0, $$rhs :> plus[drv$bayes$r2$ssa0,
1]], assignment[$$lhs :> drv$bayes$Seq$ssa0, $$rhs
:> fix[drv$bayes$r3$ssa0]], assignment[$$lhs :>
drv$bayes$Matrix$ssa0, $$rhs :> rand[4, 20]],
assignment[$$lhs :> drv$bayes$priorProbability$ssa0,
$$rhs :> FromDigits[RealDigits[1.0*^-4]]],
assignment[$$lhs :> drv$bayes$score$ssa0, $$rhs :>
bayes[drv$bayes$Seq$ssa0, drv$bayes$Matrix$ssa0,
drv$bayes$priorProbability$ssa0]], putret[]];
```

Figure 4.8: MAGICA Input Streams for drv_bayes and bayes Functions

The basic process of getting type and shape information from MAGICA is as follows; First, a string representation of a program must be generated based on MAGICA requirements. The front-end traverses a CFG, which passes SSA and SO transformation phase and builds an input string. An example input string is given in Figure 4.8. Then, the front-end initializes MAGICA, feeds the input to it, and executes the command to infer types. The inference results from MAGICA, as seen in Figure 4.9, are parsed and analyzed by the front-end to build a type map. A type map is indexed by each variable name and returns a data object, which contains inference results for the variable. There are four types of information provided by MAGICA for each variable: value range, intrinsic type, dimensionality and the size of each dimension. Although all four types of information are included in this map, only type and dimension are mandatory in output TDFs. The limited inference ability of the engine cannot provide explicit sizes of dimension sizes with static information only.

Except in cases where the root function has no user-defined function calls or

```
{drv$bayes$r1$ssa0,{Interval[{1.1102230246251565*^-
   16,0.9999999999999999}],$real,{st[1,30],2}}],
drv$bayes$r2$ssa0,{Interval[{3.3306690738754696*^-
   16,2.9999999999999996}],$real,{st[1,30],2}}],
drv$bayes$r3$ssa0,{Interval[{1.0000000000000004,3.9
   999999999999996}],$real,{st[1,30],2}}],
drv$bayes$Seq$ssa0,{Interval[{1,3}],$byte,{st[1,30],2}}],
drv$bayes$Matrix$ssa0,{Interval[{1.1102230246251565
   *^-16,0.9999999999999999}],$real,{st[4,20],2}}],
drv$bayes$priorProbability$ssa0,{1.0*^-
   4,$real,{st[1,1],2}}],
drv$bayes$score$ssa0,{Interval[{0,1.0*^-
   4}],$real,{st[1,30],2}}],{Times[Complex[1, 1],
Interval[{DirectedInfinity[-
   1],DirectedInfinity[1]}]],$complex,{st[putret[]],rk[putret[]]
   }}}
```

Figure 4.9: MAGICA Output Stream for drv_bayes Function

function inlining has removed function calls, the front-end must run the above process separately for every callee function. MAGICA does not provide inference results for variables in callee functions. When the front-end sends MAGICA the command to perform inference to MAGICA, a function name is given to MAGICA. MAGICA only gives inferred information for variables in this function. MAGICA performs inter-procedural type inference, assigns actual arguments to formal arguments, conducts inference algorithms within callee functions, and returns the return variable back to the caller function. But these callee functions are not exactly the target of inference, and their inference results are not included in output. For the front-end to generate TDFs for callee functions as well, we have to re-run MAGICA to infer each callee function. Input argument initialization is performed by inserting dummy assignments, which assign actual arguments to corresponding formal arguments. The pseudo-code algorithms for this process are given in Figure 4.10. The front-end recursively traverses each CFG, starting from the root CFG, and whenever it encounters a user-defined function call, invokes MAGICA to process the callee function with the dummy assignments added at the head of program. In this way, every call instance of a function will go through MAGICA and have generated its own type map.

4.G Constant Propagation

Constant propagation is one of the common high-level optimizations that eliminate unnecessary computation and register uses by replacing variables whose values can be resolved in compile time with their actual computed values (Aho et al., 1986; Kennedy and Allen, 2002). This can open possibilities for other optimizations such as dead code elimination and common subexpression elimination. Constant propagation is usually performed in the early stages of compilation, since it is a strictly machine-independent optimization.

The main reason to perform constant propagation after conducting type inference is that we could leverage inference results from MAGICA to identify candidates for constant propagation and get their off-line computation results as well. The inference engine internally performs a sort of constant propagation to provide value/value range information for each variable. For example, the value range for variable `x` in `x=3*10` is given as 30 as a result of type inference. Therefore, we don't have to run separate fixed-point propagation algorithms typically required for constant propagation. Variables whose inferred value information is a constant value rather than a value range, are identified as candidates, and the optimization process is completed by replacing all the occurrences of candidate variables with corresponding values. This optimization is one of the example optimizations which can be simplified and accelerated by using information obtained from type and dimension inference phase.

4.H IR Generation

The last phase of the MATLAB front-end is IR generation. It traverses the CFGs for each function and generates a PCODE IR per CFG. TDFs are made from type maps built by the inference engine. There may be multiple TDFs for the same function, because type maps are generated every time the function is called with input arguments for different types and dimensions. If a function is called multiple times but with arguments with the same type and dimension, TDF will be generated only once.

A PCODE for bayes function is presented in Figure 4.11. You can see that MATLAB statements are expressed as low-level instructions such as add, mul, and beq, or as pseudo-instructions, such as call, putret, and alloc, which require additional processing in later stages to be transformed into machine instructions. High-level semantics like loops and conditions would be used by back-ends for code analysis, but won't appear in final machine instructions. Figure 4.12 shows a fragment of a TDF for bayes function. TDFs generated from different type maps are distinguished by the unique postfix in the file name. This postfixes capture the type and dimension information of return variables and input arguments. For example, a postfix of `i2_f2f1` describes that this is a type map that occurs when the function takes a float matrix for the first argument, a float scalar for the second argument, and returns one integer scalar on function termination. In the figure, you can see from the file name and the function name that this version of TDF for bayes function takes three arguments: integer matrix, float matrix, and float scalar respectively, and returns a float matrix.

```
for all functions f,
    generate_magica_input(f, true, null);
Infer_type(root_function, root_function.name());

function generate_magica_input(function f, bool root, List actualArguments) {
    if (root) {
            generate Magica input w/o actual argument assignments
    }
    else {
            for all formal argument fm in f.formal() and actual argument sa in actualArguments,
                        generate an assignment (fm = a) and insert it at the start of the code
            generate Magica input
    }
}

function infer_type(function f, string unique_name) {
    infer type for function f
    construct a type map t from MAGICA inference result
    store it to the type map of f as <unique_name, t> pair
    traverse(f.getCFG());
}

function traverse(node n) {
    if (n is a function call) {
            if (n is a Magica-supported built-in call or native function call) return;
            else {
                        callee = n.getCalleeFunction();
                        generate_magica_input(callee, false, n.getActualArguments());
                        // example: "i2_f2f1" = one integer 2D array return var, float 2D and scalar args
                        unique_name = f.getUniqueName(n.getReturnVar(), n.getActualArguments());
            }
            infer_type(callee, unique_name);
    }
}
```

Figure 4.10: Pseudo Code for Recursive Mechanisms for Callee Function Inference

```
<root>
<CFG Label="bayes">
...
<Region ID="bayes_1_list_1" Type="LOOP">
   <Region ID="bayes_1_list_1_preheader" Type="NODE">
     <Inst op="move"> <Def ID="n_ssa0"/> <Use Int="1"/> </Inst>
   </Region>
   <Region ID="bayes_1_list_1_cond" Type="NODE">
     <Inst op="phi"> <Def ID="n_ssa0_phi"/> <Use ID="n_ssa0"/> <Use ID="n_ssa0_incr"/> </Inst>
     <Inst op="bgt"> <Use ID="n_ssa0_phi"/> <Use ID="lm_ssa0"/> </Inst>
   </Region>
   <Region ID="bayes_1_list_1_list" Type="LIST">
      <Region ID="bayes_1_list_1_list_0" Type="NODE">
         <Inst op="add"> <Def ID="s4_ssa0"/> <Use ID="k_ssa0"/> <Use ID="n_ssa0_phi"/> </Inst>
         <Inst op="subsref"> <Def ID="nt_ssa0"/> <Use ID="Seq"/> <Use ID="s4_ssa0"/> </Inst>
         <Inst op="gt"><Def ID="t1_ssa0"/><Use ID="nt_ssa0"/><Use Int="0"/> </Inst>
         <Inst op="lt"> <Def ID="t2_ssa0"/> <Use ID="nt_ssa0"/> <Use Int="5"/> </Inst>
         <Inst op="and"> <Def ID="t3_ssa0"/> <Use ID="t1_ssa0"/> <Use ID="t2_ssa0"/> </Inst>
   </Region>
   <Region ID="bayes_1_list_1_list_1" Type="IF">
      <Region ID="bayes_1_list_1_list_1_cond" Type="NODE">
         <Inst op="bne"> <Use ID="t3_ssa0"/> <Use Int="0"/> </Inst>
       </Region>
```

Figure 4.11: bayes.pcode

```
<root>
 <function name="_bayes_f2_i2f2f1">
 ...
 <var ID="Matrix">
      <type> FLOAT </type>
      <shape dimension="2">
      </shape>
 </var>
 <var ID="Pa_ssa0">
      <type> FLOAT </type>
      <shape dimension="1"/>
 </var>
 <var ID="Pb1_ssa0">
      <type> FLOAT </type>
      <shape dimension="2">
      </shape>
 </var>
<var ID="Seq">
      <type> INTEGER </type>
      <shape dimension="2">
      </shape>
 </var>
      <var ID="lm_ssa0">
      <type> INTEGER </type>
      <shape dimension="1"/>
 </var>
```

Figure 4.12: bayes_f2_i2f2f1.tdf

Chapter 5

Performance Evaluation

In this chapter, we evaluate the performance of the MATLAB front-end in terms of the execution time of generated machine instructions. We will examine a total of four versions of the front-end, one baseline version without any optimizations, and three optimized versions with different optimization configurations. The primary goals of performance evaluation are (1) to prove that the baseline implementation of the front-end can successfully compile a range of benchmarks into correctly-working binaries, and (2) to examine how effective front-end optimizations, i.e., function inlining and constant propagation, are in improving the efficiency of compiler output.

5.A Evalution Environment

The experimental environment is composed of three major components: compiler infrastructure, benchmarks, and target architecture/simulator. In this section, we will discuss in detail how each component is configured for the thesis.

5.A.1 Compiler Infrastructure

Though the thesis focuses only on a front-end which is portable and efficient, the general efficiency of a compiler is typically measured by its ability to collect performance numbers from instructions generated by the compiler. Therefore, a complete compiler infrastructure, including a back-end, is necessary. Therefore, we combine the MATLAB front-end with the RAW back-end, as presented in Chapter 2. To isolate performance improvement of the front-end, we use a simplified version of the RAW back-end for all experiments. If front-end IRs go though multi-threaded parallelization in the back-end, the way front-end optimizations affect performance can vary depending on the external factors. For example, some code can achieve much larger performance gain by function inlining, since its code reveals more parallelism by inlining functions, while others don't. Therefore, in this thesis, the back-end generates single-threaded instructions for one tile without performing instruction-level parallelization and optimization. Performance evaluation and analysis using a multi-threaded back-end is out of scope for this thesis, and will be included in future researches.

For the front-end, we compare the performance of one baseline version and three optimized versions. In order to confirm that the baseline front-end works well to generate correct instructions and evaluate how much optimization passes can speed up the execution based on the baseline result, we first measure the performance of the baseline front-end. This version of the front-end is focused on producing correct output by executing all the compilation stages described in Chapter 4 except for function inlining and constant propagation. There are three other versions which extend the baseline front-end by adding: (1) function inlining only; (2) constant propagation only; and (3) both respectively. The function inlining phase comes right after the AST construction phase, while constant propagation is performed after type inference. Function inlining is expected to reduce program execution time by eliminating function call overheads. More importantly, it enables the inference engine to provide more accurate and explicit type and dimension inference result, i.e., integer instead of float (more exact type) and scalar instead of array (more exact dimension), which lead to more concise and efficient instruction generation. If function inlining is an optimization which promotes effectiveness of an inference engine, constant propagation is an example of optimizations which can be facilitated by the inference engine. Its performance benefit might be marginal, but it shows that common optimizations which require variable information such as type, dimension and value can be easily built based on the engine output. The third version with both optimizations will test how constant propagation perform using more accurate inference result from the inference engine after function inlining.

5.A.2 Benchmarks

For fair performance evaluation of the compiler, we used a subset of MAT2C benchmarks and several other popular mathematical benchmarks. The MAT2C benchmark suite is a set of benchmarks used to test the MAGICA type inference engine itself (Joisha, 2003). Because it is already extensively tested and proved to work for the inference engine, we could leverage the fact to concentrate more on debugging the other parts of the front-end. Furthermore, the MAGICA project provides a set of reference input and output files for the MAT2C benchmarks, which we can refer to when validating our MAGICA input generator. These benchmarks are composed of main computation functions and a driver routine which invokes them. The ones evaluated in the thesis are: finediff, closure, crnich, editdist. Detailed descriptions of each benchmark can be found in Table 5.1.

In adddition to MAT2C benchmarks, we included more simple but widely-used mathematical benchmarks in our benchmark pool. eigen2 and lufac are inspired by MATLAB teaching codes from MIT (Department of Mathematics, 1996). They are MATLAB implementations of very common and fundamental matrix computations with moderate complexity. The codes are rewritten in part to adapt to our compilation requirements. We have two more benchmarks: bayes which implements Bayes' Rule for computing probability is presented as an example in (Joisha, 2003) and (MathWorks, 2002), while rref is an adapted MATLAB library implementation of standard algorithms to compute a reduced row-echelon form for a matrix.

The benchmarks contain a mix of various control statements, such as for, if, while, and break, as well as a vector operator(":") to access and manipulate arrays.

Table 5.1: Benchmarks used for Performance Evaluation

Benchmarks	Description	Origin
finediff	finite-difference solution to the wave equation	MAT2C
closure	transitive closure	MAT2C
crnich	Crank-Nicholson Heat Equation Solver	MAT2C
edit	edit distribution	MAT2C
rref	reduced row echelon form of a matrix	Tcode
lufac	LU-Factorization	Tcode
eigen2	Characteristic polynomial, eigenvalues, eigenvectors of a 2 by 2 matrix	Modified MATLAB rref
bayes	bayesian signal probability	MAGICA paper

The colon operator basically defines a range of values, having its starting value, increment, and ending value as its operands (Muchnick, 1997). However, it can be translated into different operations depending on the context where it's used in. When used to define induction variables for for statement, the operator returns a scalar induction variable with loop starting and ending condition information. On the other hand, the operator can also specify a range of elements when used as array indices. For example, $A(1 : 10, :) = 4$ is the same as assigning 4 to A's elements from first to 10th column in every row. In order to see the front-end handle all the possible cases with the colon operator, the benchmarks include various usages of the operator.

Each benchmark contains at least one function call outside of its driver routine. The maximum function call depth for all benchmarks is three. For the baseline front-end, the call hierarchy allows us to test if recursive function loading and type inference work well. More importantly, it enables us to examine the effect of function inlining, thereby removing function call overheads.

In order to present quantitative proof for benchmark complexity, the number of lines, the number of libraries invoked and, the total number of loop iterations in each benchmark are presented in Table 5.2. Libraries shown here include MATLAB built-in functions, as well as compiler-internal functions that implement basic binary and unary operations between arrays or between scalar and array. Operations only involving scalars are directly converted to corresponding machine instructions, so they do not require function calls. If a library function is invoked in multiple contexts with arguments of different types and dimensions, each instance is counted as an independent function. The total number of loop iterations is calculated by adding loop iteration counts of each loop in benchmarks. If loops are nested, iteration counts of nested loops are multiplied to the iteration count of the outermost loop. This helps us to grasp actual execution load of each benchmark, which is not necessarily determined by the number of lines.

Table 5.2: Benchmark Complexity (measured by # of instructions and # of library function calls)

Benchmarks	# of Instructions	# of Libraries Invoked	# of Total Loop Iterations
finediff	120	38	81
closure	62	25	18
crnich	134	47	90
edit	82	15	440
rref	79	26	5
lufac	101	47	27
eigen2	124	27	0
bayes	65	10	280

5.A.3 Target Architecture Simulator

The target tiled architecture of our compiler, RAW, provides a validated cycle-accurate simulator (Taylor et al., 2004). We use the BTL RAW simulator version 2 beta 165 with toolchain version 21 to run the tests. As the back-end is currently designed to run only single-threaded instructions, we configured the simulator to have a single tile. The simulator supports a variety of C built-in library functions, which is a feature added to help C to RAW compilation. Some of our current MATLAB libraries are implemented by calling these native C libraries for faster execution.

5.B Evaluation Results

5.B.1 Function Inlining

Figure 5.1 presents the execution cycle time of the benchmarks compiled by: (1) the baseline version; and (2) the version with function inlining. For a majority of the benchmarks, benchmark performance is noticeably improved with the inlining pass. On average, the execution time of the benchmarks decreased by 2.38x when inlining optimization was applied; crnich is improved most by 5.6x, and closure least by 1.02x.

Table 5.3: Elapsed Cycle Time of the Benchmarks Compiled by the Baseline Version and the Inliner-added Version

Benchmarks	Baseline version	Inliner version
finediff	323343	137223
closure	124259	120781
crnich	847294	150719
edit	1596023	1051654
lufac	82254	74864
bayes	369847	104031
eigen2	10651	4929
rref	92791	53950

We reduced function call overheads to be one of the direct and expected causes of performance improvement. Because our benchmarks all include one or more user-defined function calls in their driver routines, eliminating the overheads involved with the calls will cause reduced execution time. This overhead includes time and resources for bookkeeping function call stacks and return address stacks, and saving and restoring caller and callee registers. Therefore, the function execution time improves as user defined functions are invoked and overheads are removed. The case with crnich supports this observation. It shows exceptionally high speedup than the others. crnich is the only of our benchmarks which invokes another user-defined function within a loop. As seen in Figure 5.2, the crnich function calls tridiagonal, which is another user-defined function, inside the nested for loop. tridiagonal is not a trivial function. It takes four

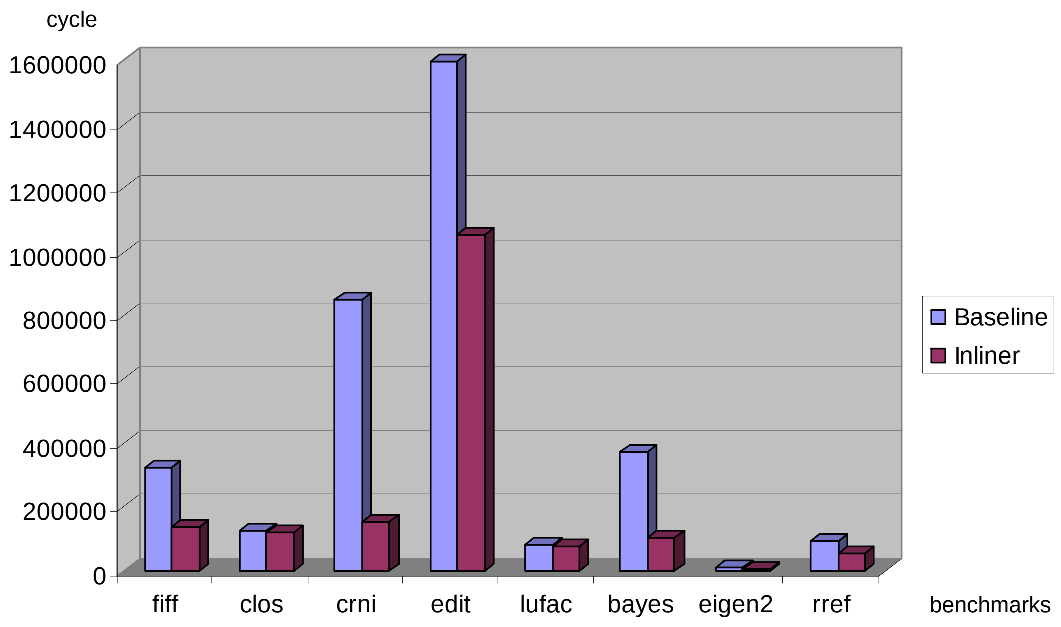


Figure 5.1: Execution time of the baseline compiler and the inlining version

arguments, performs a decent amount of computation in itself, and returns a computed array. Removing function call overhead for textsftridiagonal once will help reduce execution time. Here, function call instances are repeated as many times as the number of iterations of the outer loop. Therefore, the actual number of function calls made in run-time will be much larger than they appear in the source code. By removing all of the calls, we could achieve a huge performance boost.

On the other hand, the function inlining pass influences the quality of static type inference, which follows directly after the pass. This additional optimization effect is unique to a compiler, which depends entirely on static type inference for type information in compile-time. If the type inference engine fails to infer the explicit size of each dimension of a scalar variable, which should be 1 by 1, it provides the variable's dimension in symbolic terms using other variables' dimensions, and treats the variable as a 2D array. This is a safe assumption, because in broad sense a scalar is equivalent

```
function U = crnich(a,b,c,n,m)
...
for j1=2:m,
   for i1=2:(n-1),
      Vb(i1)=U(i1-1,j1-1)-U(i1+1,j1-1)+s2*U(i1,j1-1);
   end;
   X = tridiagonal(Va, Vd, Vc, Vb);
   U(1:n, j1) = ctranspose(X);
end;

function X = tridiagonal(A,D,C,B)
n = size(B,2);
for k=2:n,
   mult = A(k-1)./D(k-1);
   D(k)=D(k)-mult*C(k-1);
   B(k)=B(k)-mult*B(k-1);
end;
X(n)=B(n)./D(n);
for k=(n-1):-1:1,
   X(k)=(B(k)-C(k)*X(k+1))./D(k);
end;
```

Figure 5.2: Code Snippet of crinich

to a 1 by 1 array in MATLAB. The performance issue here is that the compiler must invoke a special function implementation of basic arithmetic operations, which takes arrays as arguments if a scalar is conservatively inferred as an array. That is, a+b cannot be translated into a single add instruction if either of the terms is inferred as an array. Due to limited inference power of compile-time type inference, the unnecessary cost to invoke and execute these highly complicated and lengthy functions is unavoidable. This increases the rate of symbolic inference particularly in current compiler implementation, where conservative value ranges are passed as input arguments in order to sustain the reusability of compiler output.

Function inlining solves this problem by embedding function body of callee function into caller function. Because there's no argument passing between user-defined functions in this case, exact type and dimension information obtained in caller function flows into the variables that were originally in callee function. We can expect a higher type inference success rate, i.e., explicit inference rate, in the inlined function. Let's look at Figure 5.3, which compares the ratio of explicitly inferred scalar variables to total scalar variables in the baseline and the inliner version. The baseline compiler fails to infer the exact dimensions of 20% to 75% of the variables in each benchmark, while the inlining version hauls the ratio close to 100% for all the benchmarks except lufac. The increase in the ratio closely reflects the speedup ratio for each benchmark in Figure 5.1. For example, crnich shows the most significant improvement in both figures, while lufac consistently shows no visible difference. The reason why closure does not perform well with improved type inference results can be found in the fact that it does not contain many scalar computations; it would benefit from more exact type information.

Though this solution might prevent callee functions from being reused by other caller functions, as far as callee functions are short and lightweight enough as candidates of function inlining are supposed to be, it is a fair tradeoff between reusability and performance. In case that reusability of front-end IRs, the function inlining pass can be turned off. Moreover, more accurate static type and dimension information can enable other optimization passes which come later in the front-end or the back-end. Overall, the front-end with the function inlining pass improves the quality of IR in terms of performance and type inference accuracy for all the benchmarks.

5.B.2 Constant Propagation

We experiment with another popular optimization pass – constant propagation – to see how it would speed up the benchmarks. Figure 5.5 presents the evaluation result. Adding the constant propagation pass to the baseline compiler did not produce visible speed-up for any benchmark. This disappointing result can be explained by two aspects of the process: (1) array computation-intensive characteristics of the benchmarks inherently limits the number of scalar candidates for constant propagation; and (2) type and dimension information loss at the input arguments cause low inference rate in callee functions. The first and major reason comes from the source language itself. MATLAB is a language designed to efficiently perform array-based computations. Array variables

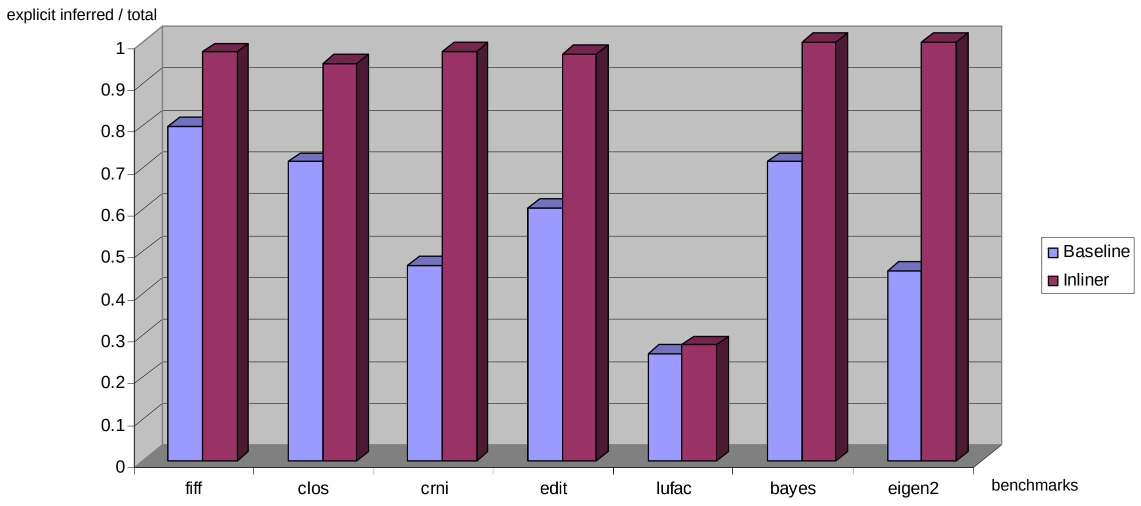


Figure 5.3: Ratio of explicitly inferred variable to the total variables (Type inference success ratio)

are not the usual target of constant propagation, while most MATLAB applications and benchmarks – including ours – are composed of array-and-array and array-and-scalar computations.

However, there exist not a few scalar variables which can be replaced with constants in each benchmark. Especially when an array is initialized by specifying its elements one by one, as in $a = [b, c; d, e]$, the compiler front-end generates many scalar variables that will compute the size of the array and the offset of each element in run-time. These computations can be significantly optimized through constant propagation, but degraded accuracy of type and dimension information for input arguments is once again an issue. In the example above, if b,c,d, or e are inferred as array with unknown sizes when they are in fact scalars, compiler-generated size and offset variables cannot be inferred correctly as scalars either. Constant propagation pass is implemented in the

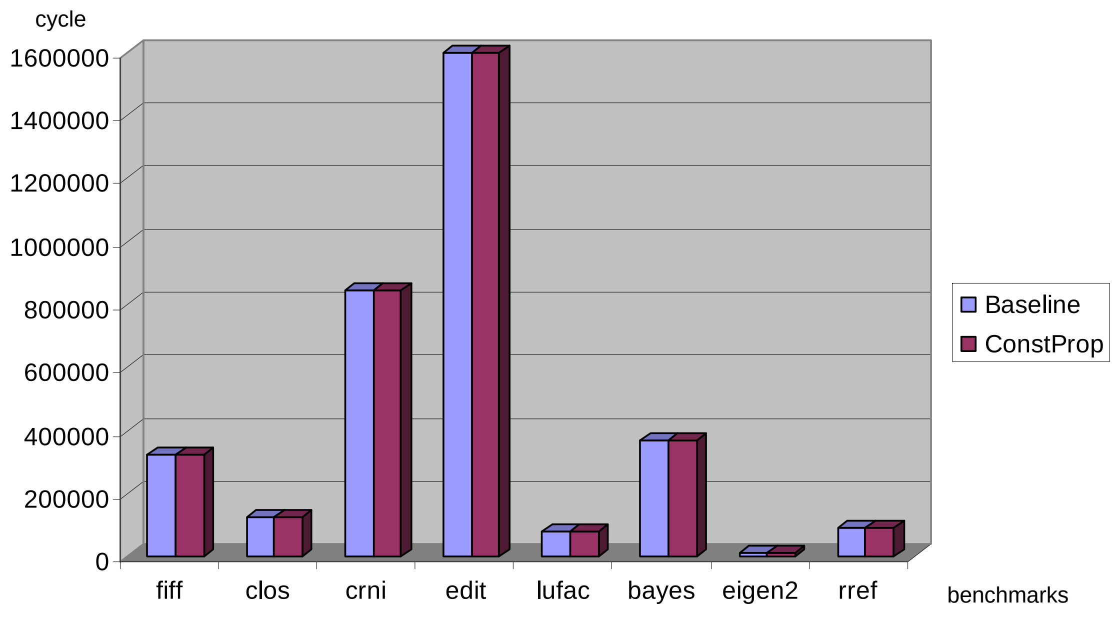


Figure 5.4: Execution Time of the Baseline Compiler and Constant Propagation Version

front-end, as it is one of the most common and verified source-level optimization passes, but it does not provide visible performance gain by its lonesome in our current compiler implementation.

5.B.3 Function Inlining and Constant Propagation

The last experiment to examine performance of different front-end optimization passes combined the two optimizations. The experimental result is shown in Figure 5.5. The figure compares the execution time of the benchmarks compiled by three different versions: the baseline compiler, the compiler with the inlining pass, and the compiler with both inlining and constant propagation passes. The result shows that most of the benchmarks perform best when they went through both the optimization passes. The inliner and constant propagation version outperforms the baseline version at most by

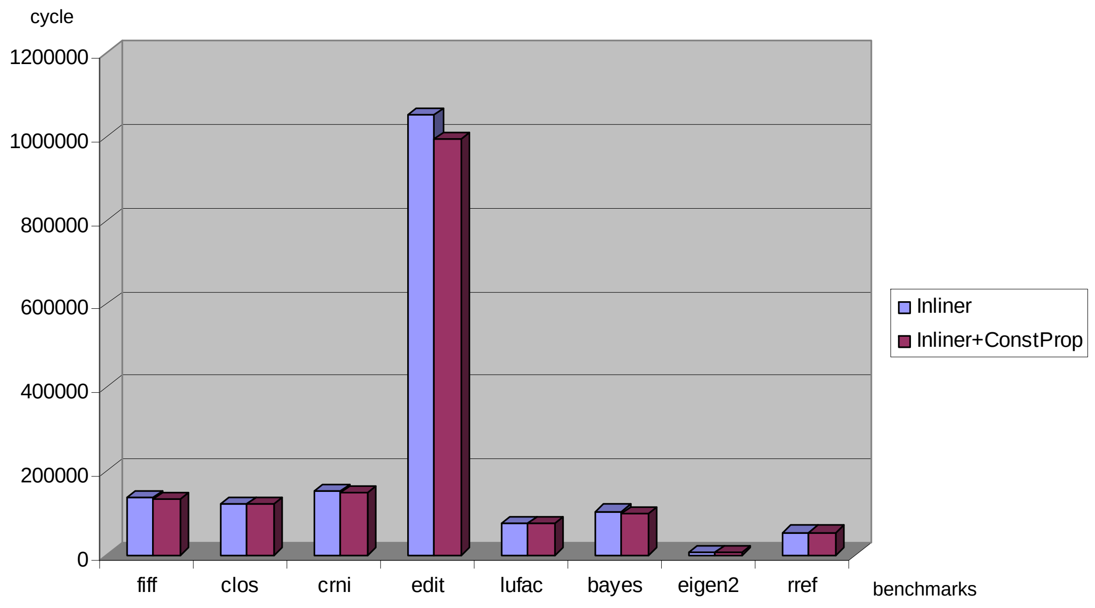


Figure 5.5: Execution Time of the Inlining Front-end and the Inlining & Constant Propagation Version

5.68x for crnich and at least by 1.03x for closure. Compared to the inliner only version, the inliner and constant propagation combined front-end produced IRs consistently faster by an average of 1.03-1.08x. It is worthwhile to remark that constant propagation altered the performance number when combined with function inlining. As function inlining eliminates type and dimension information loss around function calls and infers more scalar variables correctly, the constant propagation pass (follows function inlining) was able to have more candidates for constant propagation.

5.B.4 Summary

Table 5.4 and Figure 5.6 summarize the speed-up achieved by adding two optimization passes, function inlining and constant propagation, to the MATLAB front-end.

By eliminating user function call overheads and increasing the rate of successfully inferring exact dimensions, the function inlining pass reduced the execution time of all the benchmarks by 1.03x to 5.69x. On the other hand, constant propagation works only when there are enough scalar variables inferred as scalar constants to noticeably reduce the number of variable assignments and computations. It is observed that function inlining assits constant propagation by increasing the number of candidates for constant propagation.

The evaluation result depends largely on the design choices made for current compiler implementation. The dramatic effect of function inlining is partially due to our design decision of passing conservative value ranges for input arguments on function calls. It is also significantly affected by the efficiency of the static type inference engine. If the performance of the type inference engine itself is improved by changing the algorithms or by adding additional functionalities, the simple application of function inlining might not cause a visible speed-up although overall absolute execution time could still improve.

Table 5.4: Speedup of Three Optimized Versions to the Baseline Version

Benchmarks	Inliner version	ConstantProp version	Inliner + ConstantProp version
finediff	2.356	1.004	2.415
closure	1.029	1.000	1.035
crnich	5.622	1.000	5.689
edit	1.518	1.000	1.605
lufac	1.099	1.000	1.099
bayes	3.555	1.000	3.824
eigen2	2.161	1.009	2.300
rref	1.720	1.001	1.728

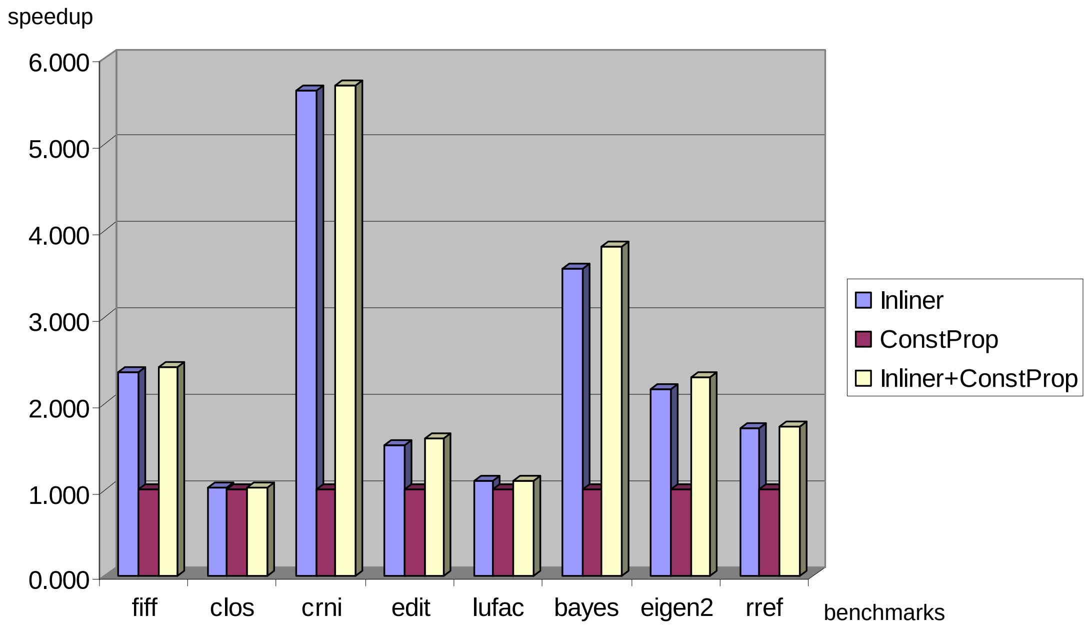


Figure 5.6: Speedup of Three Optimized Versions to the Baseline Versions

Chapter 6

Conclusion

As one of several promising multi-core microprocessors, tiled architecture's key feature is its scalable parallelizing power as the key feature. Though each tiled microprocessor is equipped with a parallelizing compiler, these compilers can not harness the full potential of underlying architecture due to insufficient parallelism in source codes. In the thesis, we proposed a new compiler front-end which will connect a popular source-base with ample parallelism to tiled architectures that can really support them.

The thesis presents a MATLAB front-end as the first implementation of a new compiler infrastructure for the class of tiled architectures. It attempts to resolve type information from typeless MATLAB sources and minimize run-time overheads by adopting a static type inference engine. Because a different implementation of the same library is linked to its callers in a MATLAB compiler depending on the type and dimension of input arguments, the libraries must be written type and dimension-aware. To make the job of populating the libraries more extensible, the front-end provides an extended MATLAB format, in which type and dimension of arguments and return variables can be explicitly forced. In terms of the entire compiler infrastructure, flexible and standardized IR between front-ends and back-ends are defined to increase the portability of each end. In the thesis, the back-end for RAW tiled architecture is paired with the MATLAB front-end for performance evaluation.

The MATLAB front-end is composed of a series of compilation phases. At first, the MATLAB parser takes MATLAB source files as input and recursively constructs AST for the root function as well as callee functions. The optimization of function inlining is performed at the AST level to eliminate user-defined function calls and embed the function bodies into the root function. The baseline compiler skips this phase. After that, the front-end constructs a CFG from an AST. A CFG consists of basic blocks, but it also includes MATLAB semantic information such as loop and conditional statements as regions around basic blocks. To facilitate various data-flow analyses and code generation, SSA and SO transformations are applied to the CFGs. The next phase is static type inference. The front-end converts the CFGs into MAGICA input streams, feeds the input programs to MAGICA, and runs the engine. To obtain type and dimension information for callee functions as well, the front-end recursively re-runs the engine for user-defined callee functions with dummy assignments, which assign initial values to input arguments. The inference result is parsed to build a type map for variables. The CFGs and type maps are printed as PCODE and TDF IRs, respectively, at the IR generation phase.

Performance evaluation was conducted on seven MATLAB benchmarks compiled by three versions of the MATLAB front-end, with a different optimization configuration. The experimental result showed that a simple function inlining pass can improve performance by 5x at most and by 2.38x on average. The increase in speed mostly comes from eliminating expensive function call overheads and enabling the static type inference engine to infer accurate dimension information for callee functions. Although the constant propagation pass was not able to reduce the execution time when applied by itself, it further speeds up most of the benchmarks by 3 to 8% on top of speedup obtained from function inlining when combined with the function inlining pass. In conclusion, evaluation results reconfirmed the importance of utilizing an efficient static type inference engine, and showed that the proper combination of optimization passes can complement the engine to a certain extent.

Important future work on the MATLAB front-end centers around improving the efficiency of static type inference and implementing more optimizations. We used an existing inference engine as a black box to sustain the complexity of the compiler, as the engine itself is very complicated. The efficiency of the inference engine, however, can be improved in many ways. The current engine often provides too broad and safe inference results when results could be narrowed down still maintaining correctness. Especially for loop variables whose values depend on the iteration count, MAGICA fails to infer the exact value range, even if it is clearly visible and guaranteed. Improving MAGICA by modifying its source codes or by adding extra processing stages for output from MAGICA will enhance the power of static inference. The latter solution can also analyze symbolic (non-constant) inference results, which are currently ignored by the front-end.

On the other hand, the current front-end contains few traditional or parallelizing optimization passes. As the SSA form enables and facilitates various optimizations, such as dead code elimination and strength reduction, optimizations can be included in the front-end to remove unnecessary computations while the back-end concentrates more on parallelizing optimizations. Front-ends that utilize other source languages will definitely help refine the IRs, which are currently tested only on the MATLAB front-end, and also expand the source-base for tiled architectures. For example, tiled architectures can be especially powerful in processing multi-media and rendering applications that include massive deadline-driven calculations. Front-ends that support streaming languages or graphics libraries will expand the accessibility of tiled architectures. Issues with back-ends are primarily focused on developing more efficient assignment/scheduling algorithms for various resources. Devising new parallelization algorithms is possible, but combining existing algorithms to find balance between coarse-grained and fine-grained parallelism can help build a more adaptable compiler to applications with different types of parallelism. As back-ends for other tiled architectures are built, and their parallelization passes are tested, new requirements might appear for front-ends, too.

Appendix A

PCODE and TDF Specification

A.A PCODE Specification

A.A.1 Tag Format

- root

  <root> <CFG> </root>

- CFG

  <CFG Label=”FunctionName”> <Region>+ </CFG>

- Region

  <Region ID=”RegionID” Type=”NODE | LOOP | LIST | IF”> <Inst>* </Region>

  Note: the topmost region should be a LIST region.

- Inst

  <Inst op=”Opcode”> <Def>* <Use>* </Inst>

  Note: each operation has different composition of Def and Use elements.

- Def

  <Def ID=”VarId”/>

- Use

  <Use ID=”VarId” | Int=”IntValue” | Float=”FloatValue” | Label=”LabelValue” | Str=”StringLiteral”/>

A.A.2 Operation Format

- add / sub / mul / ldiv / rdiv / pow

  <Inst op=”add—sub | mul | ldiv | rdiv | pow”> <Def ID> <Use ID | Int | Float> <Use ID | Int | Float> </Inst>

  Note: two-operand scalar/array arithmetic operation

- mul.e / ldiv.e / rdiv.e / pow.e

  <Inst op=”mul.e | ldiv.e | rdiv.e | pow.e”> <Def ID> <Use ID | Int | Float> <Use ID | Int | Float> </Inst>

  Note: two-operand element-wise array arithmetic operation

- and / nor / xor / or

  <Inst op=”and | nor | xor | or”> <Def ID> <Use ID> <Use ID> </Inst>

  Note: two-operand logical operation

- eq / ne / gt / ge / lt / le

  <Inst op=”eq | ne | gt | ge | lt | le”> <Def ID> <Use ID | Int | Float> <Use ID | Int | Float> </Inst>

  Note: two-operands logical comparison operation

- uminus

  <Inst op=”uminus”> <Def ID> <Use Int | Float> </Inst>

- not

  <Inst op=”not”> <Def ID> <Use ID | Int | Float> </Inst>

- move

  <Inst op=”move”> <Def ID> <Use ID | Int | Float | Str> </Inst>

- colon

  <Inst op=”colon”> <Def ID> <Use ID | Int | Float=”StartValue”> <Use ID | Int | Float=”IntervalValue”> <Use ID | Int | Float=”EndValue”> </Inst>

  Note: MATLAB colon operator assignment

- call

  <Inst op=”call”> <Use Label=”FunctionName”> <Def ID>+ <Use ID | Int | Float | Str>* </Inst>

  Note: function call(multiple return values allowed).

- putret

  <Inst op=”putret”> <Use ID | Int | Float | Str>* </Inst>

- getarg

  <Inst op=”getarg”> <Def ID>* </Inst>

  Note: argument definition

- ble / bge

  <Inst op=”ble | bge”> <Use ID | Int | Float> <Use ID | Int | Float> </Inst>

  Note: compare two operands (less then equal) and branch

- bne

  <Inst op=”bne”> <Use ID | Int | Float> </Inst>

  Note: compare one operand with 0 if they’re not equal and branch

- break

  <Inst op=”break”> </Inst>

- subsasgn

  <Inst op=”subsasgn”> <Def ID> <Use ID=”SourceArray”> <Use ID | Int | Float=”NewValue”> <Use ID | Int | Float>* </Inst>

  Note: array/matrix sub-assign (eg. A[3,4] = 5)

- subsref

  <Inst op=”subsref”> <Def ID> <Use ID=”SourceArray”> <Use ID | Int | Float>* </Inst>

  Note: array/matrix sub-reference (eg. B = A[3,4])

- phi

  <Inst op=”phi”> <Def ID> <Use ID>+ </Inst>

- alloc

  <Inst op=”alloc”> <Use Int=”NumOfDimension”> <Use Int = ”SizeOf EachDimension>+ <Def ID> <Use ID | Int | Float | Str>* </Inst>

  Note: matrix allocation (eg. C = [1,2,3 ; 4,5,6])

A.B TDF Specification

- root

  <root> <function> <variable>* </root>

- function

  <function name=”functionName”/ >

- variable

  <variable> <type=”INT | FLOAT”/> <dimension value=”NumOf Dimension” /> </variable>

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D., 1986: *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-10088-6.

Almási, G., 2001: *MaJIC: A MATLAB Just-In-Time Compiler*. Ph.D. thesis.

Almási, G., and Padua, D., 2002: Majic: compiling matlab for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 294–303. ACM, New York, NY, USA. ISBN 1-58113-463-0. doi:http://doi.acm.org/10.1145/512529.512564.

Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Bachmann, C., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., Walkden, M., and Zaretsky, D., 2000: A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, 39. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-0871-5.

Barua, R., Lee, W., Amarasinghe, S., and Agarawal, A., 2001: Compiler support for scalable and efficient memory systems. *IEEE Trans. Comput.*, **50**(11), 1234–1247. ISSN 0018-9340. doi:http://dx.doi.org/10.1109/12.966497.

Budd, T., 1988: *An APL compiler*. Springer-Verlag New York, Inc., New York, NY, USA. ISBN 0-387-96643-9.

Burke, M. G., Carini, P. R., Choi, J.-D., and Hind, M., 1995: Flow-insensitive interprocedural alias analaysis in the presence of pointers. In *LCPC '94: Proceddings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, 234–250. Springer-Verlag, London, UK.

Cheng, B.-C., and Hwu, W.-M. W., 2000: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 57–69. ACM, New York, NY, USA.

Cooper, K., and Torczon, L., 2003: *Engineering A Compiler*. Morgan Kauffman.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., 1991: Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, **13**(4), 451–490. ISSN 0164-0925.

Department of Mathematics, M. I. o. T., 1996: Matlab teaching code. http://web.mit.edu/18.06/www/Course-Info/Tcodes.html.

Etter, D., 1999: *Introduction to MATLAB*. Prentice Hall PTR, Upper Saddle River, NJ, USA. ISBN 0130131490.

Joisha, P. G., 2003: *A type inference system for MATLAB with applications to code optimization*. Ph.D. thesis, Evanston, IL, USA. Adviser-Prithviraj Banerjee.

Joisha, P. G., and Banerjee, P., 2002: Magica: A software tool for inferring types in matlab. Technical Report Technical Report CPDC-TR-2002-10-004, Department of Electrical and Computer Engineering, Northwestern University.

Joisha, P. G., and Banerjee, P., 2006: An algebraic array shape inference system for matlab®. *ACM Trans. Program. Lang. Syst.*, **28**(5), 848–907. ISSN 0164-0925. doi:http://doi.acm.org/10.1145/1152649.1152651.

Kennedy, K., and Allen, J. R., 2002: *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-286-0.

Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S., 1998: Space-time scheduling of instruction-level parallelism on a raw machine. *SIGOPS Oper. Syst. Rev.*, **32**(5), 46–57. ISSN 0163-5980. doi:http://doi.acm.org/10.1145/384265.291018.

Martin, R. C., 2003: *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA. ISBN 0135974445.

MathWorks, T., 2002: Matlab digest. http://www.mathworks.com/company/newsletters/digest/sept02/.

Moore, G. E., 2000: Cramming more components onto integrated circuits. 56–59.

Muchnick, S. S., 1997: *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-320-4.

O'Brien, K., O'Brien, K. M., Hopkins, M., Shepherd, A., and Unrau, R., 1995: Xil and yil: the intermediate languages of tobey. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, 71–82. ACM, New York, NY, USA. ISBN 0-89791-754-5. doi:http://doi.acm.org/10.1145/202529.202537.

Parr, T., 2007: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition. ISBN 0978739256.

Rose, L. D., and Padua, D., 1999: Techniques for the translation of matlab programs into fortran 90. *ACM Trans. Program. Lang. Syst.*, **21**(2), 286–323. ISSN 0164-0925. doi:http://doi.acm.org/10.1145/316686.316693.

Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W., and Moore, C. R., 2003: Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *SIGARCH Comput. Archit. News*, **31**(2), 422–433. ISSN 0163-5964. doi:http://doi.acm.org/10.1145/871656.859667.

Smith, A., Gibson, J., Maher, B., Nethercote, N., Yoder, B., Burger, D., McKinle, K. S., and Burrill, J., 2006: Compiling for edge architectures. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, 185–195. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2499-0. doi:http://dx.doi.org/10.1109/CGO.2006.10.

Swanson, S., 2006: *The wavescalar architecture*. Ph.D. thesis, Seattle, WA, USA. Adviser-Mark Oskin.

Swanson, S., Michelson, K., Schwerin, A., and Oskin, M., 2003: Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 291. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2043-X.

Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A., 2002: The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, **22**(2), 25–35. ISSN 0272-1732. doi:http://dx.doi.org/10.1109/MM.2002.997877.

Taylor, M. B., and Lee, W., 2005: Scalar operand networks. *IEEE Trans. Parallel Distrib. Syst.*, **16**(2), 145–162. ISSN 1045-9219. doi:http://dx.doi.org/10.1109/TPDS.2005.24. Member-Saman P. Amarasinghe and Member-Anant Agarwal.

Taylor, M. B., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A., 2004: Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, 2. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2143-6.