MaJIC: A Matlab Just-In-time Compiler*

George Almasi and David A. Padua

Department of Computer Science
University of Illinois at Urbana-Champaign
{galmasi,padua}@cs.uiuc.edu

Abstract. This paper describes our experience with **MaJIC**, a just-in-time compiler for MATLAB. In the recent past, several compiler projects claimed large performance improvements when processing MATLAB code. Most of these projects are static compilers suited for batch processing; **MaJIC** is a just-in-time compiler. The compilation process is transparent to the user. This impacts the modus operandi of the compiler, resulting in a few interesting analysis techniques. Our experiments with **MaJIC** indicate large speedups when compared to the interpreter, and reasonable performance when compared to static compilers.

1 Introduction

MATLAB [14], a product of Mathworks Inc., is a popular programming language and development environment for numeric applications. MATLAB is similar to FORTRAN 90: it is an imperative language that deals with vectors and matrices. Unlike FORTRAN, however, MATLAB is untyped and polymorphic: the semantic meaning of identifiers is determined by their mode of use, and the operators' meaning depends on the operands.

The main strengths of MATLAB lie both in its interactive nature, which makes it a handy exploration tool, and the richness of its precompiled libraries and toolboxes, which can be combined to solve complex problems. The trouble with MATLAB is its speed of execution. Unless the bulk of the computation is shifted into one of many built-in libraries, MATLAB pays a heavy performance penalty: two or even three orders of magnitude compared to similar Fortran code.

The purpose of our work is to eliminate MATLAB's performance lag while maintaining its interactive nature. Hence the requirement for dynamic, or just-in-time, compilation, which is used for similar purposes in other programming environments, including Smalltalk and Java. **MaJIC** acts like an interpreter: there is no need to deal with Makefiles and a lengthy compilation process, and therefore the user's work flow is not interrupted.

Just-in-time compilation has been around for awhile, but has recently gained new notoriety with the rise of Java. Before Java it was called dynamic compilation. Deutsch described a dynamic compilation system for the Smalltalk language

* This work was supported in part by NSF contract ACI98-70687 and by a graduate fellowship from Intel.

S.P. Midkiff et al. (Eds.): LCPC 2000, LNCS 2017, pp. 68–81, 2001.
© Springer-Verlag Berlin Heidelberg 2001

as early as 1984 [7]. Java benefits from JIT compilation by reduced machine instruction count per line of bytecode, optimized register usage, interprocedural analysis and, in many cases, replacement of virtual function calls with static calls. Both Java and MATLAB benefit from JIT compilation because it is possible to ascribe more precise semantic meaning to the symbols in the program, thereby reducing the potential polymorphism of the code - although the optimizations turn out to be quite different in practice.

The rest of this paper is organized as follows: we set the stage by describing related work, then we give a high-level overview of our compiler. We describe some of the performance enhancing features we implemented. Finally we present the results we have obtained so far and our conclusions and plans for the future.

2 Other Work with MATLAB

The techniques used in **MaJIC** are based on those developed for DeRose and Padua's FALCON compiler [6, 5], a MATLAB to Fortran 90 translator implemented at the University of Illinois.

MENHIR [3], developed by Francois Bodin at INRIA, is similar to FALCON: it generates code for MATLAB and exploits parallelism by using optimized runtime libraries. MENHIR's code generation is retargetable (it generates C or FORTRAN code). It also contains a type inference engine similar to FALCON's.

MATCOM [13] is a commercial MATLAB-compatible compiler family, developed by Mathtools Inc. This is a full development environment which incorporates a debugger, an editor, many optimizations, and even a JIT compiler. It is only commercially available.

Octave [8] is a freeware MATLAB implementation. Octave may not be very fast, but it is GNU freeware.

MATCH [1] is a MATLAB compiler targeted to heterogeneous architectures, such as DSP chips and FPGAs. It also uses type analysis and generates code for multiple large functional units.

MatMarks [15] and MultiMatlab [17] are projects that are somewhat similar in that they extend MATLAB to deal with a multiprocessing environment. MultiMatlab adds MPI primitives to MATLAB; MatMarks layers MATLAB on top of the Treadmarks software DSM system.

Vijay Menon, at Cornell University, is currently developing a MATLAB vectorizer [16], a tool that transforms loops into Fortran 90-like MATLAB expressions, built on top of the **MaJIC** compiler framework. We are hoping to integrate his work into the compiler soon.

3 MaJIC Software Architecture

In this section we give a short overview of **MaJIC**'s architecture and mode of operation. **MaJIC**'s interaction with the rest of the world is handled by an *interpreter* similar to MATLAB's. Since MATLAB's code is proprietary we

couldn't expand it with the JIT compiler. Therefore we built our own interpreter which is capable of handling a subset of the MATLAB language. The interpreter scans the input file, parses it into a high-level Abstract Syntax Tree, and then executes the program by traversing the tree in program order. It also contains a symbol table, an operand stack, and a function finder.

The *analyzer* performs three operations: symbol disambiguation, type inference, and optional loop dependence analysis. Analysis does not change the program tree, but rather annotates it. It is designed to be invoked at runtime; its input is in effect the current execution state of the interpreter, including the current symbol table and the operand stack.

The *code generator* generates *executable* code directly based on the AST and its annotations, and does so very fast. The code generator allocates registers, emits machine code, resolves addresses and patches forward jumps.

The *code repository* maintains order between the several versions of annotations and compiled code co-existing at a given time. Since a given piece of MATLAB code might be invoked several times, and since analysis depends heavily on the state of the interpreter at the time of invocation, there could be several sets of type annotations for the same code at the same time.

4 MaJIC Compilation Techniques

In this section we describe some of the techniques we used to speed up MATLAB execution.

4.1 Symbol Disambiguation

MATLAB symbols exclusive of keywords represent one of three things: variables, calls to built-in primitives, or calls to user functions. In the interpreter the decision is made at runtime when the symbol's AST node is executed. We call a symbol occurrence *ambiguous* if the way the program is written allows multiple meanings for it. The code box below shows a loop where the first occurrence of the symbol `i` is ambiguous, interpreted as $\sqrt{-1}$ in the first iteration and as a variable afterwards.

```
while(...),
        z = i;
        i = 1;
end
```

In most MATLAB programs, however, symbol ambiguity can be removed at compile time. In particular, symbols that represent variables can be conservatively identified by a variation of reaching definitions analysis: A symbol that has a reaching definition on *all* paths leading to it must be a variable.

The transfer function in this analysis can be set up as usual, but with a slight twist. It is defined on the set of the symbols in the program $\mathcal{S}$ and the nodes of the CFG $\mathcal{N}$ as follows:

$$f : \mathcal{S} \times \mathcal{N} \rightarrow \mathcal{S}$$

$$f(s, \texttt{stmt}) = \begin{cases} s \cup \{a\} & if\ \texttt{stmt}\ has\ the\ form\ \boxed{\texttt{a} = ...} \\ s \cup \{a\} & if\ \texttt{stmt}\ has\ the\ form\ \boxed{\texttt{a}(...) = ...} \\ \phi & if\ \texttt{stmt}\ has\ the\ form\ \boxed{... = \texttt{a}...}\ and\ \{a\} \cap s = \phi \\ s & otherwise \end{cases} \tag{1}$$

Equation 1 says, in effect, that a symbol generates a reaching definition if it appears on the left-hand side. The unusual part of this equation is its third line, which says that any symbol that cannot be proven to be a variable has the side effect of killing all reaching definitions. This is necessary because of the existence of built-in functions like `clear`, which destroys the whole symbol table.

Use-def Chain Generation: the UD chain is obtained as a side-effect of the disambiguation step. It uses the results of the same reaching definitions analysis but determines the set of definitions for each variable by calculating the set union of its reaching definitions.

4.2 Type Inference

Our representation of a MATLAB expression type lies at the base of the inference system. This representation is very close to the one described in [6]. We use the following type descriptors:

- The *intrinsic type* of the expression, such as: `real`, `integer`, `boolean`, `complex` or `string`.
- The *shape* of the expression, i.e. the maximum number of rows and columns it can have.
- The *range* of real values the expression can take [2]. We do not define the range for strings and for complex expressions.

The type analyzer is a monotonic data analysis framework [18], which requires the analyzed property to form a finite semi-lattice and all transition functions to be monotonic.

Our lattice is the Cartesian product $L_i \times L_s \times L_l$ of the three component lattices, which are defined as follows:

$L_i = \{\mathcal{I}, \bot_i, \top_i, \leq_i, \vee_i\}$, where

$\mathcal{I} = \{\bot_i, bool, int, real, cplx, strg, \top_i\}$

$\bot_i \leq_i bool \leq_i int \leq_i real \leq_i cplx \leq_i \top_i$ and $\bot_i \leq_i strg \leq_i \top_i$

$L_s = \{\mathbb{N} \times \mathbb{N}, \bot_s, \top_s, \leq_s, \vee_s\}$, where $\mathbb{N}$ is the set of natural numbers, and

$\bot_s = \{0, 0\}, \top_s = \{\infty, \infty\}$; $\{a, b\} \leq_s \{c, d\}$ iff $a \leq c$ and $b \leq d$

$L_l = \{\mathbb{R} \times \mathbb{R}, \bot_l, \top_l, \leq_l, \vee_l\}$, where $\mathbb{R}$ is the set of real numbers, and

$\bot_l = \{nan, nan\}$; $\top_l = \{-\infty, \infty\}$;

$\{a, b\} \leq_l \{c, d\}$ iff $\{a, b\} = \bot_l$ or $(c \leq a$ and $b \leq d)$

A *type calculator* implements the transition functions for all the known AST types. A fairly large number of rules are required due to the many built-in MATLAB functions.

Convergence Issues. The monotonic data analysis framework is guaranteed to converge if the lattices involved are finite. Our lattices are finite (due to the fact that real numbers are represented in finite precision), but very large and, therefore, analysis can take a very long time. A simple example is presented below. The range of x keeps expanding in steps of 1.0, and it takes an impractically large number of iterations to reach the stationary value of $+\infty$.

```
while(...),
  x = x + 1;
end
```

A similarly simple example can be made up with a shape that grows very slowly.

```
while(...),
  x = [x x];
end
```

"Runaway" types keep changing as the iterations progress, and hence are easily detectable. In these cases **MaJIC** sets the types in question to $\top$ to force convergence.

4.3 Just-in-Time Analysis

In **MaJIC** analysis is performed very late, just before the code is executed. This late timing benefits the analyzer by contributing items of information to which a static analyzer doesn't have access.

- We can assume that the files in the MATLAB search path, and hence the names of the defined user functions, are not going to change between compilation and execution. This allows for greater precision in the symbol disambiguation step. There is an underlying assumption that the MATLAB code is not self-modifying.

– The analyzer has access to the entire interpreter state at the point where the compiler is invoked, including knowledge of the exact types and values of all defined variables. This translates into knowledge of all defined types at the point of entry into the compiled code. The extra information reduces the requirement for heavy analysis because very simple techniques yield good results. For example, there is no symbolic analysis in **MaJIC**.

Constant propagation is achieved as a side-effect of range propagation. A value is constant if its range contains a single number. Because the input information is so precise, many scalar values that would be symbolic parameters in static analysis end up as constants.

Exact shapes are inferred in **MaJIC** by adding another lattice to the type inference engine. Normally, shape inference delivers only upper bounds on the shape of MATLAB arrays; we append a dual lattice L_t to calculate lower bounds too. An exact shape results where the lower and upper bounds are determined to be equal.

Induction variables' ranges can also be used to infer exact shapes. Given an array index expression like `A(x)`, the range of `x` is used to determine the shape of A. Normally, this is an upper bound since there is no guarantee that `x` actually reaches the limits of its range. An array index can only be used to determine a lower bound if it assumes its range limits at least once. In a loop without premature exit instructions, the induction variable is guaranteed to hit its range limits. This "boundary-hitting" property can be propagated through index expressions that combine induction variables and constants to determine lower bounds for the shapes of arrays.

There are a number of ways in which exact shapes can be used to achieve better performance. For example, array bound checks can be removed wherever the array index expression's upper range limit is less than the lower bound of the corresponding array shape. On the right-hand side, this removes the requirement for mandatory error testing; on the left-hand side, it removes the auto-resizing check.

MATLAB's own companion compiler, `mcc`, has command-line switches to disable subscript checks, but using this option does not guarantee that the compiled program will work correctly. **MaJIC** removes subscript checks conservatively. The extra effort for just-in-time analysis is very low, comparing favorably with more conventional techniques such as those described in [10].

4.4 Code Generation

The design of the code generator is based on the trade-off between compilation speed and the quality of generated code, as well as a desire to be able to plug in several native code generators. We chose a two-layered approach: a high-level code selector traverses the AST and chooses the appropriate native code generating routines, which then emit native instructions.

The Code Selector traverses the AST and reads the annotations associated with each node. It performs a top-down pattern matching to select the appropriate implementation for any given AST node. For example, for a multiplication node there are a number of possible implementations, including scalar multiplication, dot product, vector-matrix multiplication and complex vector multiplication, to mention just a few.

Here is a short summary of the non-trivial techniques used by the code selector.

- Preallocation of temporary arrays, when their maximal sizes are known at compile time. Because of the Fortran 90-like expression semantics, complicated MATLAB vector expressions involve temporary matrices at each step, and preallocating these eliminates many calls to `malloc()`. For example, in an expression like $a \times b \times c$, at least one temporary matrix is needed to hold the result of $a \times b$. If upper bounds are known for the shapes of a and b, this temporary can be pre-allocated.
- Elimination of temporaries. Techniques have been developed to deal with this in Fortran 90-like languages [12]. **MaJIC** takes a simple way out and performs code selection to combine several AST nodes into a single library call. For example, expressions like `a*x+b'*c` may be transformed into a single call to `dgemv`.
- When indexed with out-of-bound subscripts, MATLAB arrays attempt to resize themselves to include the offending subscript. Thus, in some loops, arrays are resized in each iteration with the associated performance penalty. We implemented a simple, but effective, strategy of allocating about 10% more memory than needed whenever an array is resized dynamically. Since dynamic resize tends to happen in loops, the surplus of memory will likely be used up later, meanwhile reducing the number of calls to `malloc()`. Speedups from this technique can reach two orders of magnitude or more, depending on the characteristics of the loop.
- Complete unrolling of vector operations when exact array shapes are known. This technique is most effective on small (2×1 to 3×3) matrices and vectors because it eliminates all loop control instructions. In the future, we are planning to expand this technique to completely unroll small MATLAB loops with exact loop bounds.

The Native Code Generator emits code into a string buffer, and then the buffer is simply executed. Native code generation is also a multi-pass process.

Unfortunately there is no really general-purpose dynamic code generator that suits our purposes. The closest contender is Dawson Engler's `vcode` [9], a dynamic assembler with its own RISC-like instruction set. `vcode` has been ported to many popular CPU architectures including sparc and mips, but the x86 port is useless because it doesn't include the floating point instructions.

`tcc` [19] is a C-like language layered on top of `vcode`. It adds a lot of syntactic sugar and low-level optimizations like peephole optimization and register allocation, all performed at runtime. We used `tcc` for our implementation.

The second choice for the **MaJIC** backend is Paolo Bonzini's `GNU Lightning` [4], a dynamic assembler with a platform-independent instruction set. It is ported to the Sparc, PPC and IA32 architectures. `GNU Lightning` is based on Ian Piumarta's `ccg` dynamic assembler. The problem with `GNU Lightning` is that in striving to be architecture independent it restricts the user to 6 integer registers and a stack-based floating point system. These restrictions degrade performance on RISC architectures.

4.5 Repository Management

Since MATLAB is inherently polymorphic, the same source code can have several analyses attached to it, depending on how it is called. The repository keeps these analyses in order and tries to match existing compiled code to any call that is made. The concept of a match is defined very simply, based on the input data used by the analyzer: a given interpreter state matches an analysis stored in the repository if all the affected symbols have a "smaller" ($\leq$) type than the ones used by the analysis. For example a function compiled for complex arguments can be reused when invoked with real actual arguments.

If a function is invoked many times with incompatible arguments, each invocation might require a new analysis, thereby diminishing performance. To increase the change of future matches, **MaJIC**'s analyzer has the option of generalizing the input types before starting processing. This, of course, also results in a performance penalty for the compiled code.

MaJIC uses a gradual strategy for changing types. The first analysis is performed with the types as they are; the second analysis discards range information and exact shapes. If a third call is performed and none of the performed analyses match, all inputs are treated as matrices; finally, if yet another analysis is needed, it is performed with no information whatsoever from the input.

In real life, most MATLAB invocations are monomorphic; none of our benchmark functions need more than two analyses.

5 Experimental Results

To gauge the performance of the **MaJIC** compiler, we measured both the speed of the generated code and the time spent compiling code. Our frame of reference for execution speed is set by the MATLAB interpreter. We measured the performance of three compilers: MATLAB's own compiler `mcc`, the FALCON compiler, and **MaJIC**.

Surprisingly, for FALCON we obtained speedup figures that were better than those reported in [6]. We attribute this to the better Fortran 90 compiler we used as well as to a CPU with a large amount L2 cache that rewards the inherently better locality of compiled code.

5.1 The Benchmarks

We ran our experiments on the benchmarks originally gathered by Luiz DeRose. These are described in [6], so we will limit ourselves to a short description. The 12 benchmarks are all numerically intensive and relatively short: between 25 and 125 lines of code. They are as follows:

- `cgopt`, `icn`, `qmr`, and `sor` are linear equation solvers.
- `crnich`, `dirich`, `finedif`, `galrkn`, `orbec`, and `orbrk` are PDE solvers: finite difference solvers, a FEM solver, and n-body solvers.
- `adapt` is an adaptive quadrature algorithm.
- `mei` is a fractal generator.

The matrices in the benchmarks have sizes between 50×50 and 450×450. The problem sizes were originally determined to yield approximately 1 minute run times, but newer computers tend to finish them faster. We ran all our experiments on a Sun workstation with Solaris 2.7 and a 270MHz Ultrasparc processor.

5.2 Performance Figures

Figure 1 shows the speedups of `mcc`, **MaJIC**, and FALCON on a logarithmic scale. These speedups are based on execution times only (i.e., they don't take into account the compilation times). Missing bars mean speedups are below 1.0.

Execution times tell only half the story, however. Figure 2 shows the time spent from one user prompt to the next in each benchmark (i.e. the time spent by the user waiting for the result). For **MaJIC** and `mcc`, we assume that the compilation time is included in the prompt-to-prompt time; `mcc` tends to be used a lot from the MATLAB prompt, and **MaJIC** incurs the compilation penalty by default. We show two sets of speedups for FALCON, with and without compilation time included.

As the figures show, **MaJIC**'s performance falls between `mcc` and FALCON when judged on execution time alone. However, **MaJIC** takes much less time to compile than `mcc`, making it a better choice when the code is edited - and thus recompiled - frequently.

We can classify the benchmarks based on which compilation techniques influence their performance the most.

- The `adapt` benchmark benefits from the dynamic over-allocation scheme mentioned in section 4.4.
- The `cgopt` and `qmr` benchmarks benefit from a better implementation of the `dgemv` routine, as they spend more than 99% of their time doing matrix-vector multiplication. `qmr` derives additional benefits from an exact match between the MATLAB code and the API of the `dgemv` function. These benchmarks do not benefit much from type analysis; we get similar performance numbers with type analysis turned off.
- The `crnich` and `orbrk` functions suffer because **MaJIC** does not currently implement function inlining.

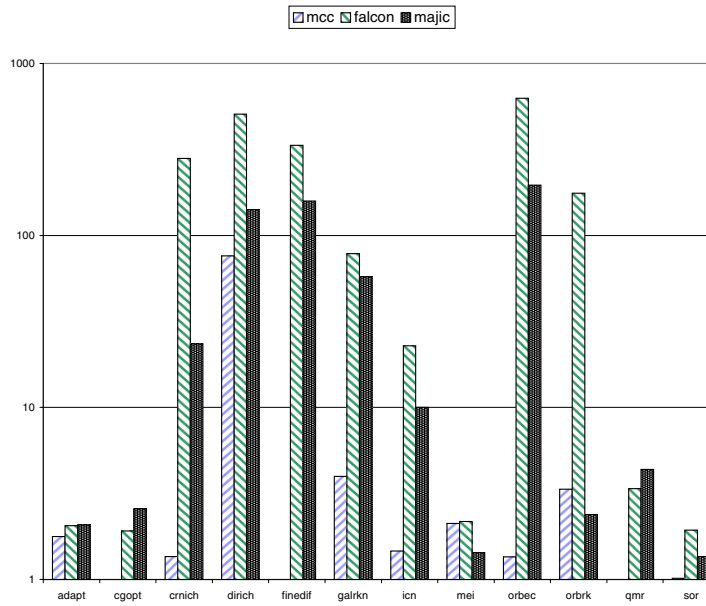

Fig. 1. speedups based on execution time only

– `crnich`, `dirich`, `finedif`, `galrkn`, and `orbec` suffer from the lack of low-level compiler optimizations in the **MaJIC** code generator (e.g., common subexpression elimination and strength reduction). We recompiled the FALCON codes with the Fortran compiler optimizations turned off (`f90 -g`). Figure 3 compares the unoptimized FALCON codes with **MaJIC** speedups. The **MaJIC** numbers include compile time. The performance numbers are pretty close to each other. Just-in-time analysis and high level code generation techniques in **MaJIC** compensate for the time lost while compiling.

We measured the effectiveness of just-in-time analysis by running the analyzer on a subset of our benchmarks, in just-in-time mode and in simulated static mode. In just-in-time, mode the analyzer had access to the interpreter state just before starting compilation; in static mode, the analyzer was cut off from the interpreter and ran with no outside information. Of 500 AST expression nodes, the just-in-time analyzer determined 402 to be scalars; by comparison, static analysis determined only 240 of them to be scalar.

Just-in-time analysis is reasonably fast. Analysis involves 2 to 20 traversals of the abstract syntax tree. The 270MHz Sparc processor that ran the benchmarks took about 0.2 milliseconds per iteration to process an AST node; average compilation time (including disambiguation, type inference, and code generation) is about 170 milliseconds.

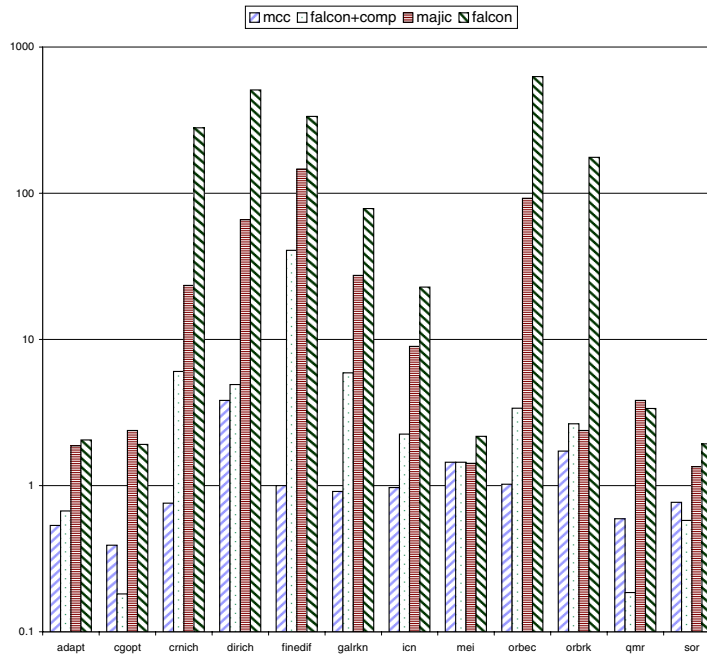

Fig. 2. speedups based on execution + compile time

Figure 4 shows the relative composition of execution time in **MaJIC**. Disambiguation and type inference never take more than 15% of the total time. Actual execution of the code dominates the time spent between user prompts. Code generation can take up to 40% of the time in the case of benchmarks that have large speedups.

6 Conclusion and Future Plans

We have built and tested a just-in-time compiler for MATLAB. It provides adequate performance while reducing the compilation overhead by at least an order of magnitude.

There is still a lot of room for improvement. The quality of the compiled code is mediocre at this point, due to the fact that the dynamic compiler/assembler combinations we use for code generation are not doing any low-level optimizations. Experiments show that by performing common subexpression elimination on the dynamically generated code we could improve performance by at least anoher 50some of the benchmarks; we are planning to implement these next.

Another technique we are planning to implement is to overlap compilation with computation. One of the ways to do this is to perform hybrid compilation,

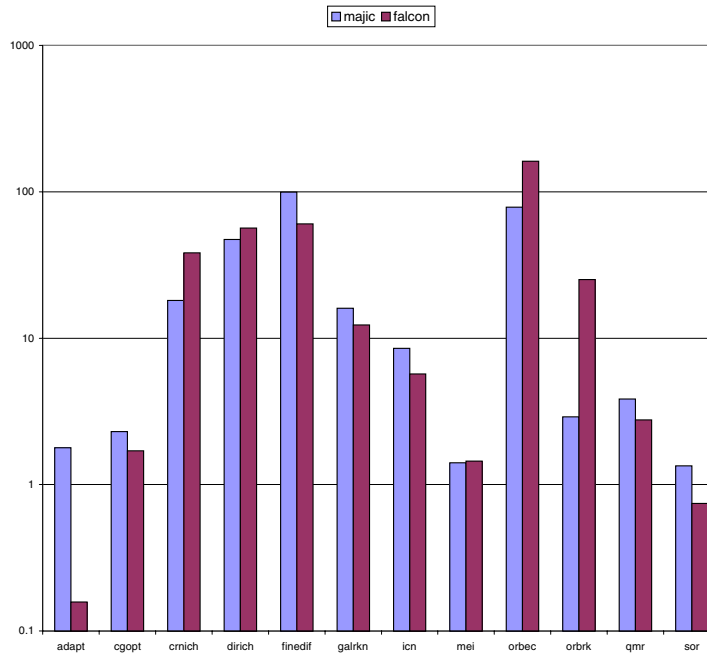

Fig. 3. Speedups without F90 optimizations

i.e. analyze and compile as much of the code in advance as soon as it is available, and to generate semi-compiled templates that can be quickly refined by the JIT compiler.

We also have other techniques under consideration. Some of the MATLAB libraries can be replaced by better ones [20, 11]; some of the loops can be executed in parallel on SMP machines. FALCON performance figures show that function inlining is a very cost-effective technique.

References

[1] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. Match: A matlab compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Center for Parallel and Distributed Computing, Northwestern University, Aug. 1999.

[2] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[3] F. Bodin. MENHIR: High performance code generation for MATLAB. `http://www.irisa.fr/caps/PEOPLE/Francois/`.

[4] P. Bonzini. The GNU Smalltalk Homepage.

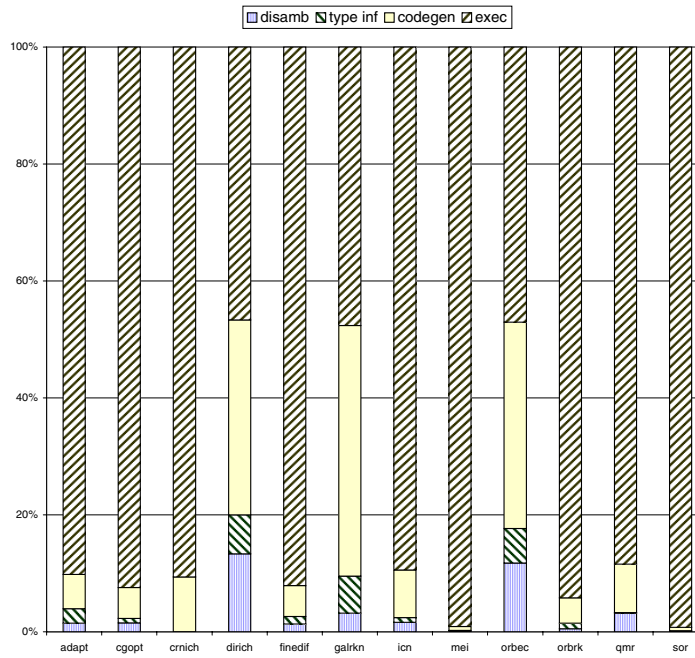

Fig. 4. Composition of execution time in **MaJIC**

[5] L. DeRose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):285–322, March 1999.

[6] L. A. DeRose. Compiler Techniques for MATLAB Programs. Technical Report UIUCDCS-R-96-1996, Department of Computer Science, University of Illinois, 1996.

[7] L. P. Deutsch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[8] J. W. Eaton. Octave homepage. `http://bevo.che.wisc.edu/octave/`.

[9] D. R. Engler. VCODE: a portable, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '96)*, Philadelphia PA, May 1996.

[10] R. Gupta. Optimizing array bounds checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, 1993.

[11] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–753, November 1997.

[12] G.-H. Hwang and J. K. Lee. An array operation synthesis scheme to optimize fortran 90 programs. In *PPOPP '95*, pages 112–122, Santa Clara, USA, 1995.

[13] Mathtools inc homepage. `www.mathtools.com`.

[14] Mathworks Inc. homepage. `www.mathworks.com`.

[15] Matmarks homepage. `http://polaris.cs.uiuc.edu/matmarks/matmarks.html`.

[16] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *1999 ACM Conference on Supercomputing*. ACM SIGARCH, June 1999.

[17] V. Menon and A. E. Trefethen. MultiMATLAB: Integrating MATLAB with high-performance parallel computing. In *Proceedings of Supercomputing '97*, 1997.

[18] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall, 1981.

[19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, May 1997.

[20] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, 1998.