Library Function Selection in Compiling Octave

Daniel McFarlin* Arun Chauhan
{dmcfarli,achauhan}@cs.indiana.edu
Department of Computer Science, Indiana University, Bloomington, IN 47405 USA

Abstract

One way to address the continuing performance problem of high-level domain-specific languages, such as Octave or MATLAB, is to compile them to a relatively lower level language for which good compilers are available. As a first step in this direction, specializing the high-level operations in the source, based on operand types, leads to significant gains. However, simple translation of the high-level operations to the underlying libraries can often miss important opportunities to improve performance. This paper presents a global algorithm to select functions from a target library, utilizing the semantics of the operations as well as the platform-specific performance characteristics of the library. Making use of the library properties, the simple and easy-to-implement selection algorithm is able to achieve as much as three times performance improvement for certain linear algebra kernels, over a straight mapping of operations, which are compiled to the vendor-tuned BLAS.

1. Motivation

The ease of programming offered by *High-level Domain-specific Languages*, also called *scripting languages*, has resulted in their growing popularity in the recent years. Examples of such languages include MATLAB® (and its open-source version, Octave), Perl, Python, S-Plus (and its open-source version, R), and PHP.

One way to address the performance problem that continues to hinder large scale application development in these languages is to *compile* these languages to a relatively lower-level language for which there are known good compilers [7, 11, 3]. Usually, the first step in this translation process is inferring types of the variables from their definitions and uses so that the individual high-level operations in the original program could be mapped to their more precise equivalent operations. For example, in a hypothetical high-level language, an operation such as "+" could refer to string concatenation or arithmetic addition, depending on the context. Static resolution of the operation, leading to type-based specialization, can result in impressive performance gains [3].

However, these gains can be limited by the actual mapping of the high-level source operations to those available in the underlying target language. For some contexts the mapping is obvious—if the operands to a source operation are proved to be numerical scalar values there may be a simple equivalent scalar operation in the target language that can perform that operation. On the other hand, if the source operation has no equivalent primitive in the target language then the compiler must either explicitly generate code to implement the high-level source operation or seek an appropriate library routine that implements that operation. For example, this is the case when an operation in Octave, or MATLAB®, is found to involve arrays and the target language, such as C, does not support array operations primitively.

This paper develops an algorithm to map the high-level operations in Octave to a given library, in a language such as C or C++, which implements those operations. A large number of array operations encountered in real Octave programs can be directly mapped to the BLAS, which is the test-bed used in this paper. Identifying the platform specific characteristics of the library, and incorporating their knowledge into the code generator, can result in up to factor of three speed improvement, in some cases, over a code generator that ignores library characteristics and performs a direct mapping of the high-level operations to the underlying library.

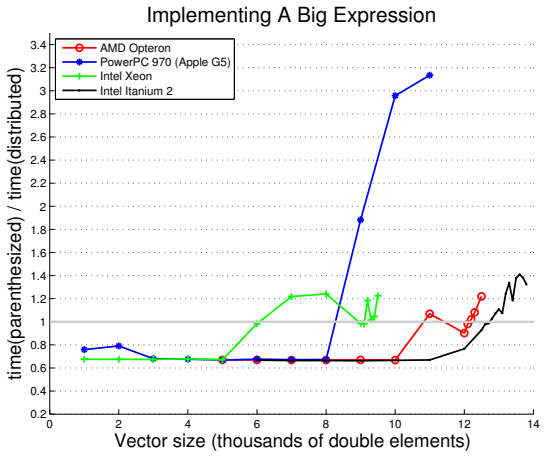
2. Selecting the Library Functions

At first sight, mapping the high-level Octave operations to an underlying library, or libraries, appears to be an *instruction-selection* problem [5]. However, several characteristics set the problem apart from standard instruction-selection.

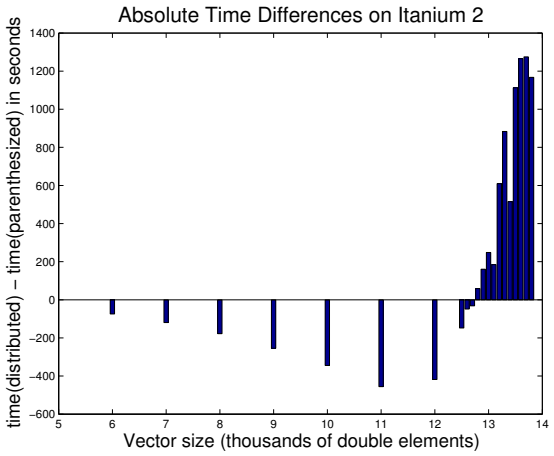
- Since each "instruction" is really a library call, its cost

*The work in this paper was supported in part by the National Science Foundation under Grant No. EIA-0116050, and a grant by the Lilly Endowment. Daniel McFarlin was partly supported by Hutton Honors College Summer Research grant and the Indiana University Department of Cognitive Science Summer Research grant.

®MATLAB is the registered trademark of The MathWorks Inc. Octave is an open-source language that is syntactically compatible with MATLAB. The paper discusses Octave, but the techniques also apply to MATLAB almost without change, unless noted otherwise.



(a) Performance ratios.



(b) Absolute gains.

Figure 1. Minimizing the number of multiplications is not always a win.

may be much harder to compute than the simple instructions for a RISC, or even CISC. The cost of the library operations is dependent on a number of variables and is dynamic in nature, specific to the call-site context, and may depend on the run-time environment in complex ways.

- Multiple alternatives are likely to be available to perform a single operation, from multiple libraries, from multiple functions within a library, or even as explicit code generated by the compiler using lower-level libraries or primitive operations in the target language. The choices are usually many more than those offered by machine instruction sets.
- The penalty of making a mistake can be much higher since arrays might be involved. A bad choice can easily result in unnecessary memory copies leading to highly inefficient execution.

Consider the following Octave expression:

$$\mathtt{A} + \mathtt{A} * \mathtt{B}' + 2 * (\mathtt{A} + \mathtt{B})' * \mathtt{A} + (\mathtt{x} + \mathtt{y}) * \mathtt{x}' \qquad (1)$$

In order to translate this code into C or C++ the first step is inferring the variable types. The type of a variable is defined as the pair (τ, ψ) where τ is the primitive type (such as `real`, `boolean`, `complex`, etc.), and ψ is the size for array variables (including the number of dimensions and the extent along each dimension). Various other pieces of work have addressed the issue of inferring types in MATLAB, and equivalently Octave, and that will not be discussed further in this paper [7, 2, 11]. The work described here assumes the availability of the variable types.

The type inference pass might deduce that A and B are square matrices and x and y are vectors of the same length as the number of rows and columns in the matrices. Assuming that the numerical considerations allow arithmetic commutativity and distributivity, pure algebraic considerations favor evaluating the expression in its existing form than trying to distribute the multiplication over addition (i.e., "opening" the parentheses). If the multiplication is completely distributed the expression becomes:

$$\mathtt{A} + \mathtt{A} * \mathtt{B}' + 2 * \mathtt{A}' * \mathtt{A} + 2 * \mathtt{B}' * \mathtt{A} + \mathtt{x} * \mathtt{x}' + \mathtt{y} * \mathtt{x}' \qquad (2)$$

The original expression (1) performs two full matrix products and one vector outer product, while the distributed expression (2) performs three full matrix products and two vector outer products. The number of additions remains unchanged, although the former adds fewer elements because one of the additions involves vectors. Most analytical models that operate with linear algebra will prefer the first version. In fact, this example is taken from a published study that made exactly that choice [9].

Figure 1(a) shows the ratio of times taken by version (1) to version (2) on four different processors with increasing data sizes. Somewhat counter-intuitively, there is significant performance gain implementing the second version of the expression beyond a critical matrix-size. The exact size that determines the transition is platform dependent. The dramatic saving in time is clearer when the absolute gain is plotted, as the Figure 1(b) shows for Itanium 2. Since the version (2) performs better for larger sizes the gains it produces are much more significant than the gains resulting from the version (1) when it is more efficient. Similar dramatic gains exist on other platforms as well, especially PowerPC.

In order to understand the reason behind this difference consider the final sequence of BLAS functions that is used to implement each version of the expression, shown below using an abstract notation. The original expression (1) is implemented with the sequence on the left and the expression (2) maps to the sequence on the right.

```
copy(A,tmp0);
gemm(1,A,B,1,tmp0);
copy(A,tmp1);
axpy(1,B,1,tmp1);
gemm(2,tmp1,A,1,tmp0);
copy(x,tmp1);
axpy(1,y,1,tmp1);
ger(1,tmp1,x,tmp0);
```

```
gemm(1,A,B,1,tmp0);
ger(1,x,x,tmp0);
ger(1,y,x,tmp0);
gemm(2,A,A,1,tmp0);
gemm(2,B,A,1,tmp0);
```

The second sequence uses more level 3 BLAS calls, but eliminates three matrix copies. This results in reduced memory bandwidth requirement, especially for larger data sizes that do not fit in cache. A compiler cannot realize this tradeoff by restricting itself to the mathematical properties of linear algebra. In the case of linear algebra operations utilizing the BLAS the compiler needs to know the tradeoffs of the various functions on the target platform. The rest of the paper develops a *library function selection* algorithm and evaluates it on the BLAS.

3. Mapping Operations

A simple approach to translating Octave code would be to map each individual operation to an equivalent library call. For instance, the Octave statement

```
A = B*C;
```

could be translated to a call to `DGEMM` assuming that the compiler has been able to determine that `B` and `C` are matrices of `real` values. Additionally, the compiler must also insert memory management code for allocating the array `A` and deallocating arrays `B` and `C` if these will not be used subsequently.

If the matrices `B` and `C` were column and row vectors, respectively, the call to `DGEMM` would still be valid but is no longer the most efficient implementation for the Octave statement. Figure 2 shows the comparative times of vector outer-product using two different `BLAS` functions, `DGER` and `DGEMM`. It is clear that it is better to use `DGER` on all platforms for all vector sizes that were tested. Interestingly, on Intel Xeon, `DGEMM` is as fast as `DGER` and, therefore, the second level of specialization yields little additional benefit.

In general, there may be multiple libraries, or even multiple functions within a single library, available to perform an operation. In traditional instruction selection, especially for CISC where similarly multiple instructions may be available to implement an operation, such cases have been handled using simple analytical cost models for the instructions. When the individual "instructions" are library calls building analytical cost models and ensuring the accuracy of those models is a challenge. The problem is exacerbated by the rapidly evolving and complex hardware technology that results in complex tradeoffs that can be difficult to predict. Moreover, while detailed analytical models might be available for the well-studied BLAS, the same may not hold for other libraries.

Our approach is to drive the instruction selection using empirical data whenever there is a choice. We have found that in most cases, empirical evaluation tends to consistently favor one over the other. In general, the relative costs of multiple equivalent implementations of a high-level operation may

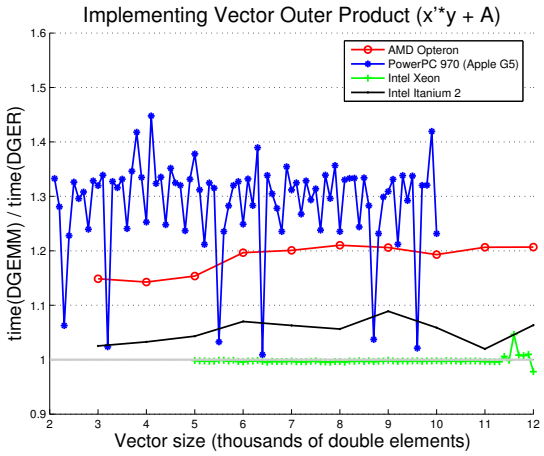


Figure 2. `DGEMM` and `DGER` can both perform vector outer-product.

depend on the context, including operand sizes, cache conflicts, available memory bandwidth, etc. In such situations, more will need to done to pick the best choice based on the compile-time knowledge. Our current implementation handles contexts based on array sizes, but disregards the effects of cache conflicts and memory bandwidth. Extending the implementation to handle more such contextual parameters is part of the planned future work.

To summarize, the first step in effective function selection for a target library is to evaluate the available implementations for an operation:

Step 1: Build a table of relative costs (currently done by empirical measurements) for different contexts (currently, array sizes) for all the available implementations of an operator, to be used by `function-selector`.

Even though the table can help select an appropriate function for a single operation, when considering a slightly larger context than one single operation the optimal choice may be dictated by other factors, as the motivating example showed in Section 1 and which is also the topic of discussion of the next section.

4. Statements in Basic Block

As in most other languages, all complex expressions in Octave must be broken down into binary operations and computed one operation at a time. In order to infer the types of all the intermediate values and disambiguating the heavily overloaded operations the compiler first *flattens* all the expressions into the most basic operations that can be directly mapped to either primitives or available library functions in the target language. This creates explicit names for all the intermediate values. For example the Octave code `R = a + b*C` will get *flattened* to `t1 = b*c; R = a + t1;`, where `t1` is

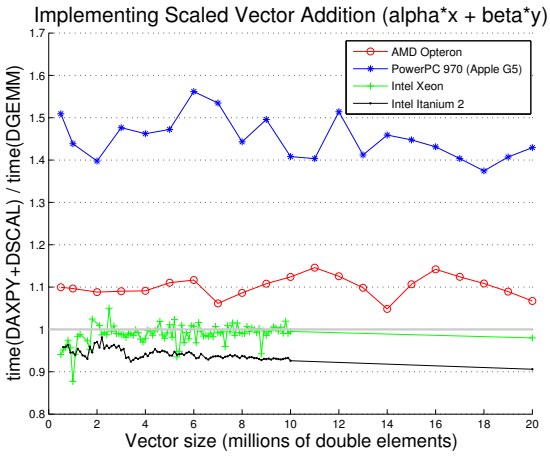


Figure 3. `DGEMM` **is better than scaling+addition.**

a compiler-generated uniquely named temporary. Further, the program is converted into the Static Single Assignment (SSA) form to aid analysis [6]. We use a novel algorithm that operates directly on the Octave Abstract Syntax Tree (AST) without having to build a control-flow graph [4].

The completely flattened SSA form of the program is amenable to simple operation mapping outlined in Section 3. However, such a simple approach misses several optimization opportunities. Consider the following sequence of statements on the right and a commonly occurring Octave statement on the left that might flatten into such a sequence:

```
A = a*V + b*W;   ⇒   t1 = a*V;
                     t2 = b*W;
                     A = t1 + t2;
```

Assuming that `a` and `b` are `real` scalars and `V` and `W` are `real` arrays, the above code performs scaled element-wise addition of two arrays. A simple translation of the code sequence on the right to use the BLAS will translate to two calls to `DSCAL` (scaling a vector) and one call to `DAXPY` (addition of two vectors) . A call to `DSCAL` can be saved by folding one of the scaling operations into `DAXPY` as it is capable of scaling one of its input vectors. Interestingly, however, `DGEMM` is also a candidate for performing the operation—it computes $\alpha * A * B + \beta * C$. Setting one of the matrices `A` or `B` to identity results in scaled matrix addition. Figure 3 shows the performance comparison of implementing the above code sequence using a single call to `DGEMM` with that using `DSCAL` and `DAXPY`. The figure shows that it is better to use `DGEMM`, *except* on Itanium 2, emphasizing the need for the compiler to be aware of platform-specific properties of the target library.

The above example also brings up another important aspect of library function selection: it is possible to fold multiple simple operations into a single library call. One way to think about this is *forward substitution*. In this case the expressions computing `t1` and `t2` could be forward substituted into the third statement to obtain the statement on the left. Even though this process appears to argue against flattening, there are multiple reasons why complete expression flattening is still desirable:

1. The original program may never have an expression in the form that maps directly to the target library. Flattening followed by forward substitution can build such expressions where none existed in the source.

2. Flattening makes all the temporaries needed in the program explicit, greatly facilitating the process of type inference.

3. Since complex expressions must eventually be evaluated one operation at a time, flattening simplifies code generation.

This is similar to the three- or four-address intermediate representations commonly found in traditional compilers. In our case, however, an operation may have arbitrary number of operands and results.

Not every operation may be implemented using a library call that is capable of "absorbing" other operations. For example no other operation can be combined into array scaling, using the BLAS call `DSCAL`. We call the library function that can perform multiple Octave operations in a single call a **multi-op** function. For example, both `DGEMM` and `DAXPY` are multi-op functions. An operation that maps to a multi-op function is called a **candidate** operation.

This leads to the second step in the function-selection process:

Step 2: Map the flattened statements to the target library, using the `function-selector` algorithm.

Figure 4(a) shows the algorithm `function-selector` based on the above ideas. It works on a basic block. At the end of applying the algorithm there may be dead code left corresponding to statements that compute values that have already been subsumed by multi-op functions.

Step 3: Perform dead-code elimination to remove subsumed operations.

One possible source of non-optimality in the above algorithm comes from ignoring the issue of library function scheduling (similar to instruction scheduling) that might affect the choice of the function calls. A second source is from selecting an "optimal" library function for a single operation and then folding multiple operations into it—the choice might change if the two steps were combined.

5. Global Algorithm

The algorithm in Figure 4(a) works on basic blocks, but is easily extended beyond by observing that if an operand was

defined in a ϕ-function then its value could be defined in multiple ways depending on the control-flow. If any operand of a candidate operation comes from a ϕ-function, the control-flow structure causing the ϕ-function is replicated around the statement involving the candidate operation and the algorithm recursively invoked on the new construct. Figure 4(b) outlines the changes that must be made to the text that appears on grey background in Figure 4(a) to get the final algorithm and Figure 5 illustrates it with an example.

```
algorithm basic-block-function-selector
inputs: P = Octave source code (as AST)
        S = SSA graph of P
        L = target library
outputs: R = Modified version of P with operations
             mapped to the function calls in L
             whenever possible
---------------------------------------------------
set R to an empty AST
for each simple statement, s, in P, do
  if (s does not have an operation implemented by L)
    add s unchanged to R
  else
    let ⊗ be the operation in s
    let ω be the optimal choice of function in L
      implementing ⊗ based on the current context
    if (ω is a multi-op function and
        ⊗ is a candidate operation)
      for each operand u
        let d be the statement defining u, obtained from S
        if (d can be subsumed in ω)
          add the operands of d to ω
        endif
      endfor
    endif
    add the call to ω to R
  endif
endfor
```

(a) Function selector for basic blocks.

```
let d be the statement defining u, obtained from S
if (d is defined by a φ-function and
      the φ-function has no incoming back-edges)
  replicate s into the control structure for φ (see Figure 5)
  rename the left hand sides of s uniquely
  insert a φ-function at the end of the new construct
  call function-selector on the new construct
else if (d can be subsumed in ω)
  add the operands of d to ω
endif
```

(b) Changes to the basic-block version to obtain the global function selector.

Figure 4. The `function-selector` algorithm.

The global algorithm follows the SSA graph and is linear in the size of the SSA graph and the source program. However, each candidate operation can lead to replication. The number of times a statement is replicated is bounded by the arity of the multi-op library function being considered. In the worst case each operand, a, for a candidate operation could potentially come from a ϕ-function, ϕ_a, causing replication of the statement `s` by a factor of $|\phi_a|$, the number of arguments to ϕ_a. Therefore, the total time to process a statement with candidate operation is bounded by $\prod_{a \in \text{operands}(s)} |\phi_a|$. These steps are repeated for each simple statement, `s`, in the program (including those enclosed within compound statements), resulting in the total algorithmic complexity given by the following expression:

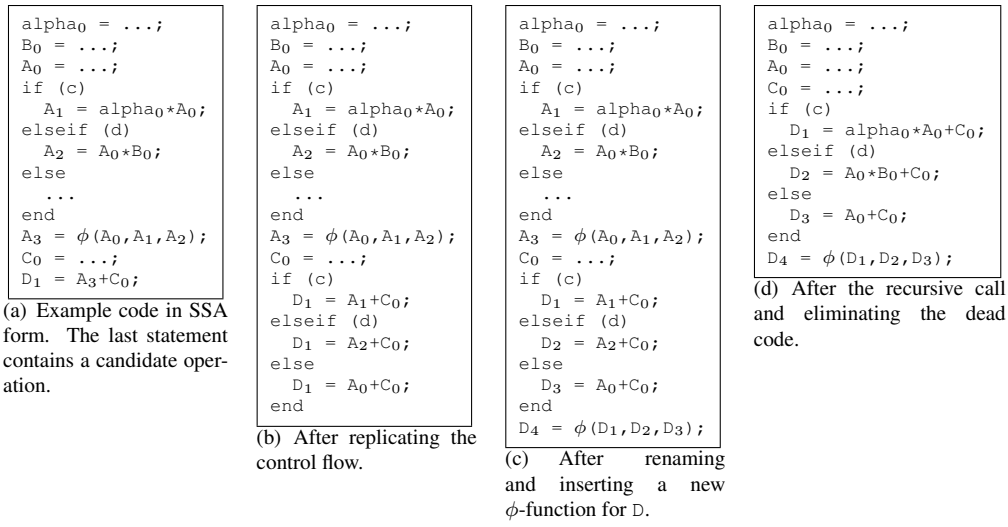
$$\sum_{s \in \text{simple-stmt}(P)} \prod_{a \in \text{operands}(s)} |\phi_a| \tag{3}$$

There is one final factor, which is the result of considering all permutations of the operands for forward substitution (assuming the operation is commutative, as is the case with many linear algebra operations). Fortunately, this only results in a small constant factor in most cases. The BLAS routine `gemm`, which has the highest arity, has six operands, resulting in an upper limit of 128 on the factor. However, only a small fraction of those 128 permutations are viable.

Notice that cascaded code replication, caused by nested recursive calls, is avoided as long as a multi-op function is itself not a candidate for absorption into another multi-op function. The above expression for time complexity looks large, but is really almost always linear in the size of the program for two reasons: The in-degree of ϕ-functions is bounded by a small constant for structured flow graphs such as those encountered in Octave programs, and the SSA graphs are usually quite sparse [6]. In all our experiments the running time of the algorithm was insignificant, and often immeasurably small, relative to the running times of the applications.

6. Experimental Evaluation

We evaluated the algorithm on five different linear algebra application kernels written in Octave, on three different platforms. Wherever applicable, the programs were compiled to 64-bit binaries. Vendor-optimized compilers and BLAS libraries were used on all platforms. All compilers were invoked with the -O3 optimization flag. All test programs were written in **C**. Timing was captured by surrounding the BLAS routines with `gettimeofday` calls. Code outside the main loop was not included in the timing measurement, because it contributed insignificantly to the total running time. Timing results were obtained from 100 runs using arithmetic mean, which was not significantly different from the median. Figure 6 compares the performance with straightforward translation that performs no forward substitution, but does minimize array allocation and deallocation across loop iterations by reusing space whenever possible. The graphs are plotted for the entire range of data sizes that could fit in memory, but the data sizes have been normalized to bring out the trends clearly.



(a) Example code in SSA form. The last statement contains a candidate operation.

(b) After replicating the control flow.

(c) After renaming and inserting a new ϕ-function for D.

(d) After the recursive call and eliminating the dead code.

Figure 5. Example illustrating the global function selection algorithm.

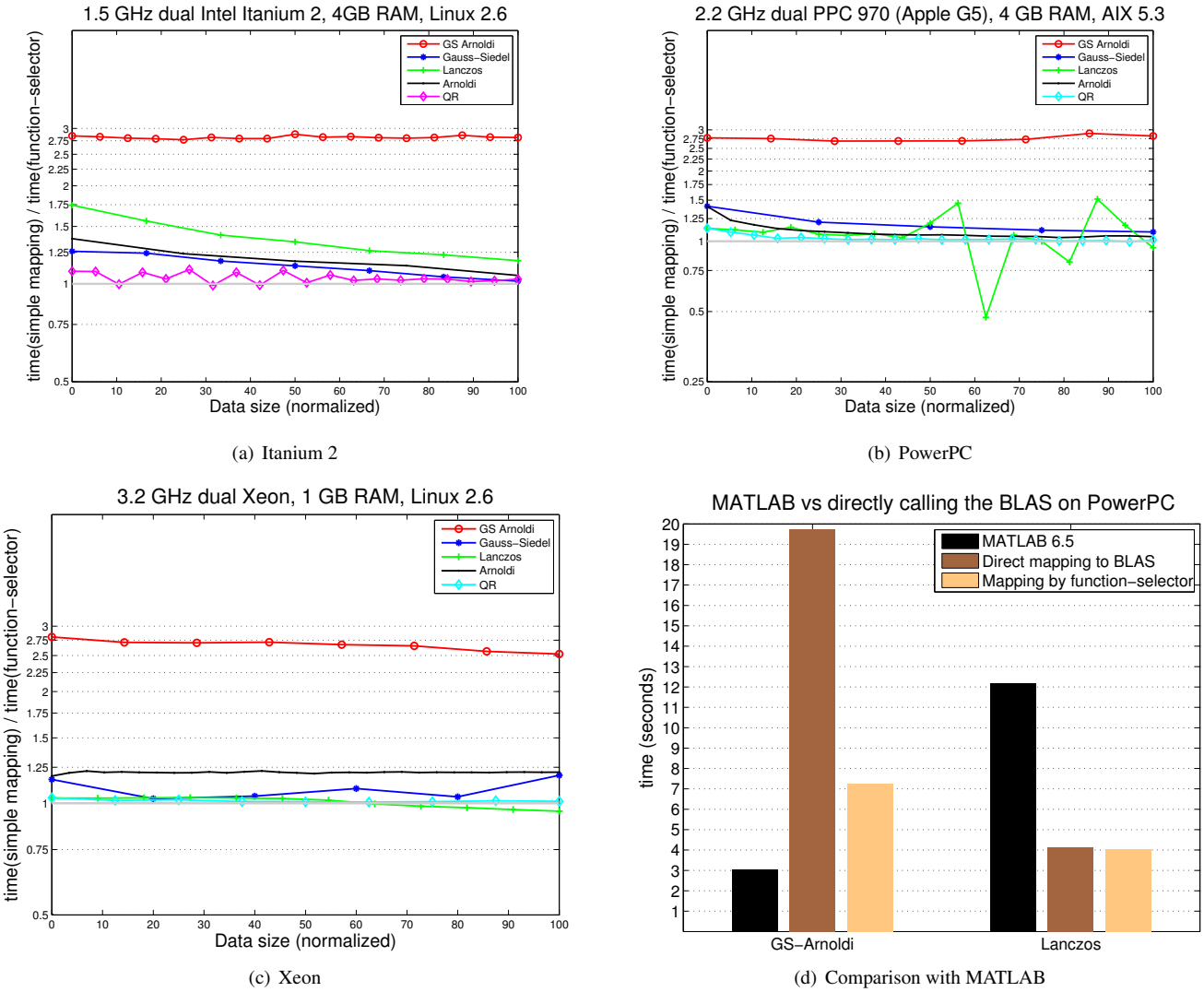
The benefits of the algorithms vary across applications, but stay relatively consistent across platforms. The kernel `GS-Arnoldi` benefits the most. It is a small piece of code with one tight vector scaling loop. The function selection algorithm is able to map the scaling particularly well onto the BLAS `DAXPY` call, while the direct mapping of individual operations introduces copies inside the tight loop, causing high overheads. There are significant gains on almost all other kernels as well, although those are less dramatic than `GS-Arnoldi`. Some of the kernels, especially, `Arnoldi` and `Lanczone`, show a reduction in gains with larger data sizes. A possible reason is that the improvements in the code generated by the `function-selector` come mainly from reduced memory operations. As the data sizes increase, the computational cost increases more rapidly than the size, amortizing the data copying costs.

Finally, the bar graph shows a comparison of running times on MATLAB, with the C code generated by a simple operation mapping and the code generated by `function-selector`, for two of the kernels for representative data sizes. For `GS-Arnoldi` MATLAB does better than directly calling the BLAS, because the computation is simple enough (repeated vector scaling and accumulation) that writing a tight loop, even in MATLAB, can eventually outperform any heavyweight BLAS function. Interpretation through byte-codes and a Just-In-Time compiler enable MATLAB to overcome the parsing overheads in the loop. This clearly shows that the compiler must consider the alternative of directly generating loops. Previous studies have also encountered similar cases [12]. Finally, even though `function-selector` does little better than a simple operation mapping on `Lanczos`, directly calling the BLAS greatly improves the performance over MATLAB. In both cases Octave is an order of magnitude or more slower than MATLAB.

7. Related Work

APL was, perhaps, the earliest array processing language and the pioneering work done on compiling APL is still highly relevant to modern high-level languages [1]. Several important transformations identified in the early studies on APL, such as *beating and dragging-along*, are still being explored in newer contexts [14]. The idea behind beating and dragging-along is to perform operations in terms of the original copy of an array as much as possible, instead of creating intermediate temporaries—for example, when operating on a subsection or the transpose of the array. The `function-selector` algorithm implicitly performs a limited form of *beating and dragging-along* whenever that can be absorbed within the multi-op functions. This is the case with transposition, for example. On the other hand, beating and dragging-along array subsections is not an obvious win on modern processors with hierarchical memories.

Other recent attempts at compiling MATLAB have largely focused on type inference and specialization, rather than efficient generation of calls to the underlying libraries [7, 11, 3]. However, it is worth comparing our approach with the techniques that were explored in an extension of the FALCON system at the University of Illinois [7, 12]. We do not have access to a working copy of FALCON to make direct comparisons to our work, but published documents indicate that the FALCON compiler attempted to match the BLAS calls to linear algebra operations in MATLAB. However, their algorithms stopped at basic block levels, whereas the algorithm presented here operates globally. Moreover, our strategy is targeted at a class of libraries, rather than being BLAS-specific. FALCON made use of a special rewriting mini-language and was limited by its power. Our use of a much more powerful tree manipulation toolkit allows us to perform much more sophisticated transformations [15]. Finally, FALCON used no empirical or



(a) Itanium 2

(b) PowerPC

(c) Xeon

(d) Comparison with MATLAB

Figure 6. Evaluation of `function-selector`. Data sizes (horizontal axis) have been normalized so that 100 represents the maximum data sizes that fit in memory.

analytical models for function selection, greatly limiting its applicability to general libraries. While there is an ongoing debate about the relative benefits of analytical and empirical models we have found experimentally determined data to be highly useful in our selection algorithm [16].

Traditional techniques for instruction selection, including those based on peep-hole optimization and tree matching, usually operate on basic blocks. In Octave, basic blocks tend to be very short and restricting library function-selection to basic blocks could seriously limit the optimization potential of function-selection. By operating beyond basic blocks, the algorithm presented here is able to overcome some of those limitations. Even when extended beyond basic blocks, the standard techniques are limited by the analysis available at the relatively low level of the intermediate code on which those techniques operate. In generating library function calls, the Octave compiler operates at a much higher level, invariably with much smaller amount of code, and is able to make use of results from deeper analyses, such as the SSA graph.

SPIRAL is a sophisticated system to generate code for DSP algorithms from formulas written in the Signal Processing Language (SPL) [13]. The code generator presented here is a much simpler system, but shares with SPIRAL the use of a declarative framework to encapsulate domain knowledge. This makes it possible to enhance the system using the lessons learned from SPIRAL. DSP compilers have to generally deal with complex instruction sets. Most solutions proposed for DSP code generation, including instruction selection, attempt to map the problem to an optimization problem and then solve that optimization problem [10, 8]. This invariably makes the

worst-case complexity exponential. Our function selection algorithm offers a simple, highly efficient, solution that has been shown to work well for the BLAS in practice.

8. Concluding Remarks

This paper developed an algorithm for function-selection for mapping high-level Octave operations to a target library. In order to get a good mapping of the original program the compiler must examine larger contexts and map *sequences* of high-level operations to suitable library functions.

Most of the past work in this direction has been restricted to basic-block optimization. The global algorithm presented here is driven by context-sensitive empirical data for a target library on the target hardware and works on the SSA graph of the program.

The contributions of this paper include:

1. A global efficient algorithm for function call selection to implement high-level operations of Octave that is shown to be effective for practical linear algebra code. The simplicity of the algorithm makes it easy to implement in a practical compiler.

2. Empirical evidence in support of the merit of unconventional mappings of linear algebra operations to the BLAS (Figure 3).

3. Empirical evidence for the BLAS that indicates that optimizations based on purely abstract properties of library functions are insufficient, sometimes even counterproductive (Figure 1).

Going back to the motivating example of Section 1, there is a choice between distributing multiplication across addition at the cost of higher number of multiplications and keeping the expression parenthesized at the cost of more memory copies. That transformation is beyond the scope of the library function-selection work presented here. However, the `function-selector` algorithm achieves minimal memory copies if an earlier optimization phase does decide on distributing the multiplication.

References

[1] P. S. Abrams. *An APL Machine*. PhD thesis, Stanford Linear Accelerator Center, Stanford University, 1970.

[2] A. Chauhan and K. Kennedy. Slice-hoisting for array-size inference in MATLAB. In *16th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[3] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *Proceedings of the ACM / IEEE SC Conference on High Performance Networking and Computing*, Nov. 2003.

[4] A. Chauhan, D. McFarlin, and P. Malpani. Directly translating MATLAB abstract syntax tree to the static single assignment form. Technical Report TR642, Indiana University, Bloomington, Indiana, Dec. 2006.

[5] K. D. Cooper and L. Torczon. *Engineering A Compiler*. Morgan Kaufmann, Dec. 2003.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[7] L. DeRose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, Mar. 1999.

[8] E. Eckstein, O. König, and B. Scholz. Code instruction selection based on SSA-graphs. In *7th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, volume 2826/2003 of *Lecture Notes in Computer Science*, pages 49–65. Springer Berlin / Heidelberg, 2003.

[9] C. Gomez and T. Scott. Maple programs for generating efficient FORTRAN code for serial and vectorised machines. *Computer Physics Communications*, 115(2–3):548–562, Dec. 1998.

[10] V. Jain, S. Rele, S. Pande, and J. Ramanujam. Code restructuring for improving real time response through code speed, size trade-offs on limited memory embedded dsps. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1863/2000, pages 459–463. Springer Berlin / Heidelberg, 1999.

[11] P. G. Joisha and P. Banerjee. An algebraic array shape inference system for MATLAB. *ACM Transactions on Programming Languages and Systems*, 28(5):848–907, Sept. 2006.

[12] B. A. Marsolf. *Techniques For The Interactive Development Of Numerical Linear Algebra Libraries For Scientific Computation*. PhD thesis, University Of Illinois At Urbana-Champaign, 1997.

[13] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb. 2005. special issue on Program Generation, Optimization, and Platform Adaptation.

[14] D. J. Rosenkrantz, L. R. Mullin, and H. B. H. III. On minimizing materializations of array-valued temporaries. *ACM Transactions on Programming Languages and Systems*, 26(6):1145–1177, Nov. 2006.

[15] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, D. Batory, C. Consel, and M. Papers, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, Berlin / Heidelberg, Germany, 2004.

[16] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005. special issue on Program Generation, Optimization, and Platform Adaptation.