HiLO: High Level Optimization of FFTs *

Nick Rizzolo and David Padua

University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801-2302
{rizzolo,padua}@cs.uiuc.edu,
WWW home page: http://polaris.cs.uiuc.edu

Abstract. As computing platforms become more and more complex, the task of optimizing performance critical codes becomes more challenging. Recently, more attention has been focused on automating this optimization process by making aggressive assumptions about the algorithm. Motivated by these trends, this paper presents HiLO, the high-level optimizer for FFT codes. HiLO blends traditional optimization techniques into an optimization strategy tailored to the needs of FFT codes and outputs C code ready to be further optimized by the native compiler. It has already been shown that such high-level transformations are important to coax the native compiler into doing its job well. HiLO provides a more appropriate platform for researching these phenomena, suggests an optimization strategy that improves on previous approaches, and shows that even software pipelining at the C level can improve the final binary's performance.

1 Introduction

As computing platforms become more and more complex, the task of optimizing performance critical codes becomes more challenging. It is seldom feasible anymore to hire a software engineer who is an expert both in the algorithm that needs to be optimized and the target architecture to write assembly code. Thus, more concerted efforts are now being expended in the research community to automate this process. Automatic tuning systems such as SPIRAL [1], FFTW [2], and ATLAS [3] are devoted to the optimization of codes in a specific domain. As such, they can make assumptions about their input from information that general purpose compilers just don't have available to them, reaping significant performance benefits.

In particular, SPIRAL [1] automatically generates finely tuned DSP transforms by searching through the space of formula decompositions to find an implementation of the transform that can be tuned well on the host architecture. Once a formula decomposition is selected, its optimization and translation to

* This work was supported in part by the National Science Foundation under grants CCR 01-21401 ITR and 03-25687 ITR/NGS; by DARPA under contract NBCH30390004; and by gifts from INTEL and IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

C is carried out by the SPL compiler [4]. As shown by [4], classical optimizations (the most significant of which was array scalarization) performed at the C source level enable native compilers on several platforms to produce more efficient object code. This result is interesting because it shows that algorithm specific performance improvements can be realized in a platform independent manner. In addition, it shows that although native compiler designers are not necessarily experts in the particular domain we are interested in, their expert knowledge about their architecture is manifested in their compiler, and it can be utilized fruitfully.

Motivated by these ideas, this paper presents HiLO, the high level optimizer for FFT codes. HiLO can, in fact, be viewed as a framework for studying domain specific optimization, although it has only been tested on FFT codes thus far. It is written modularly, each optimization pass in its own source file, adhering to the concept of separation of concerns. The order in which optimization passes will be executed and any parameters those passes might take can be specified on the command line. These characteristics facilitate research concerning the optimizations' interaction as well as lending themselves toward an automatic tuning environment in which some search is being performed over the optimizations themselves.

Furthermore, HiLO is a domain specific compiler that produces C code that competes with and sometimes outperforms that produced by SPL. HiLO blends traditional optimization techniques into an optimization strategy tailored to the needs of FFT codes and outputs C code that is more conducive to further optimization by the native compiler. In addition to exposing new ways to make optimizations work better with each other, HiLO adds software pipelining to the list of optimizations that can yield performance benefits at the C source level. The addition of software pipelining with the right parameters to a pre-existing HiLO optimization strategy that had been applied to all FFT formulas of size 32 yielded an execution time for the best behaved formula that was 35% faster than the time of the previous best formula. This is not to be misconstrued as a 35% improvement on the state of the art, but instead a very good indicator that high level software pipelining is a promising direction for further study.

The rest of this paper is organized as follows: Section 2 discusses some of the decisions we made while implementing HiLO. Section 3 mentions the analysis techniques employed in our compiler. Section 4 talks about how we tailored our optimization passes to the needs of FFT codes and our overall optimization strategy. In section 5, we describe our experimental setup and results. Finally, in section 6, we conclude and discuss future research directions.

2 Implementation Considerations

HiLO was designed with the goal of easing the development process and bolstering the research process. Its design is organized by separation of concerns, and its accepted language includes only what is necessary. Its internal representation and optimization passes are implemented in Java. Java was chosen because it

is object oriented, it garbage collects, and its extensive data structure and algorithm libraries seem more standardized and user friendly than `C++`'s STL. In addition, stack traces including line number information make debugging easier.

2.1 Command Line Interface

HiLO's command line options are used to control the selection, ordering, and parameters of optimization algorithms, none of which are executed by default. Thus, the entire machinery of the compiler is exposed both to the researcher who can easily automate his experiments, and to the automatic tuning system that incorporates HiLO as its back-end compiler and can then include those parameters in its search space. In addition, there is an argument used to include any number of optimization passes in a loop that continually applies them in order until none make any changes to the internal representation of the program in an entire pass through the loop. This utility proved quite useful in discovering the symbiotic nature of certain optimizations, as described further in section 4.7. HiLO's output is C code sent to standard output.

2.2 Front End and Accepted Language

HiLO's front end is implemented with the automatic scanner generator JLex[1] and the automatic parser generator CUP[2]. The language that it accepts is a small subset of `C`. Only assignment statements, return statements, `for` and `while` loops, and function calls are supported. Function calls are assumed to be calls to external functions that don't have side effects. The `int` and `double` data types and single dimensional arrays of those are supported. Pointers are allowed as arguments to functions and are assumed to point to arrays in mutually exclusive memory at run-time, freeing def-use and dependence analysis to compute more precise information. Any compiler directives in the input are preserved in the output, but are otherwise ignored.

This setup is specifically designed to accept as input the C code that the SPL compiler produces as output. This code is iterative, and it operates on an array of size $2n$ interpreted as n complex values. SPL and its companion FFT formula generator program `splse` are used to provide HiLO with all of the codes it optimizes. The appropriate command line parameters are given to SPL so that its internal optimizations are turned off.

2.3 Internal Representation

HiLO represents a source program internally with a simple abstract syntax tree (AST). The visitor pattern [5] enables the detachment of the algorithms that traverse the AST and operate on it from the code that implements the AST. Thus, debugging of an optimization pass is confined to a single source file.

[1] JLex was written by C. Scott Ananian.
http://www.cs.princeton.edu/∼appel/modern/java/JLex

[2] CUP was written by Scott E. Hudson.
http://www.cs.princeton.edu/∼appel/modern/java/CUP

3 Analysis

As shown in [4], the handling of arrays is one of the biggest shortcomings of general purpose compilers. When arrays cannot be converted to scalars, care must be taken in the analysis of FFT codes to glean as much information as possible from subscripted array references. Fortunately, the subscripts in FFT codes are always simple linear functions of the induction variables of enclosing loops. As such, the heavy duty analysis in HiLO is based on induction variable recognition. There is nothing unusual about HiLO's induction variable recognition; see [6] for a description of it. Building from that point, HiLO's dependence and def-use analyses can be as precise as needed. Below are descriptions of how these two analyses have been engineered to suit FFT codes. For introductions to them, again, see [6].

3.1 Dependence Analysis

As mentioned above, the subscripts in FFT codes are always simple linear functions of induction variables. Furthermore, the references to any given array within the scope of a specific nested loop have subscripts that differ only by the coefficient on the induction variable of the immediately enclosing loop and the constant if they differ at all. In addition, it is frequently the case that most indices of a given array are written to no more than once in the entire program. When an index is written to more than once, it almost always happens that each write is either in different loops or in the same loop with precisely the same subscript expression. Finally, and most importantly, HiLO's software pipelining algorithm requires only same-iteration dependence information.

Thus, a very obvious dependence test is sufficient to prove the independence of every pair of memory accesses that are in fact independent within a given loop iteration. HiLO simply checks to see if there is any single integer value that the induction variable of the immediately enclosing loop can take that will make a given pair of subscripts equivalent.

3.2 Def-use Analysis

Links in the def-use and use-def chains of subscripted variables are established whenever the GCD dependence test[3] fails to prove independence. We have not witnessed any SPL produced FFT codes in which this weak test doesn't prove sufficiently strong.

4 Optimization

We have implemented the following optimization passes in HiLO: array scalarization, algebraic simplification, common subexpression elimination (CSE), constant and copy propagation, dead code elimination, induction variable elimination, induction variable strength reduction, loop unrolling, register renaming,

[3] Again, see [6] for a description of the GCD dependence test.

basic block scheduling, and software pipelining. Those that have a significant impact on FFT or that have been tailored for FFT in some way are discussed in turn below.

4.1 Array Scalarization

The more loops are unrolled, the bigger benefit array scalarization will have on the generated code. Of course, when loops are fully unrolled, the subscripts in FFT codes all turn to constants, and then entire arrays can be replaced. But HiLO also checks for any definition of a subscripted variable for which all uses are defined exclusively by that definition. In such cases, the defined subscripted variable and all its uses can be replaced with the same scalar.[4]

4.2 Algebraic Simplification

Standard algebraic simplifications including constant folding and combining additive operators (for example, $-x + y \rightarrow y - x$) need to be performed, mainly to support the efforts of the other optimization passes. For instance, constant folding can create new constant and copy propagation opportunities.

Another important goal that this pass achieves is the canonicalization of constants to non-negative values. This technique was shown to improve performance in [7], and we have seen the same improvement. When a negative constant is discovered, it is translated to a unary negation operator applied to a positive constant. Unary operators can then combine with additive operators in the surrounding context and be simplified away. Since the majority of constants in FFT codes appear in both their positive and negative forms, this optimization nearly cuts both the size of the compiled program's constant table and the number of constant loads performed by the compiled program in half.

Expressions can also be canonicalized in a similar fashion. Expressions of the form $-x - y$ can be translated to $-(x + y)$. In a multiplicative context, unary negation is expanded to contain the entire multiplicative operation rather than just one of its factors (i.e., $(-x) * y \rightarrow -(x * y)$). Along with constant canonicalization, these translations reduce the number of forms a given expression can be represented in, thus creating previously unavailable opportunities for CSE to further simplify the code. The translation $(-x) * y \rightarrow -(x * y)$ is also useful when combined with the augmented behavior of copy propagation, discussed next.

4.3 Common Subexpression Elimination

When allowed free reign over the AST, CSE often has a detrimental affect on codes with loops. That's because, as mentioned in section 3, the majority of the expressions in subscripts turn out to be common. Pulling these expressions out

[4] If the array in question happens to be passed to the function as an argument, then HiLO will choose not to make this replacement unless it can also be proven that the given definition is not the last assignment to that index in the array.

into separate assignments to newly declared variables makes dependence analysis more complicated, and tends to add to the native compiler's confusion. With this in mind, HiLO has been endowed with a command line parameter that instructs CSE not to search for common subexpressions inside subscripts.

4.4 Copy Propagation

Copy propagation replaces occurrences of the variable on the left hand side of a given "simple" assignment statement with the right hand side of that assignment statement. In the context of FFT optimization, we consider an assignment statement simple when its left hand side is comprised of either a scalar variable or a subscripted array reference with either a constant or scalar subscript, and its right hand side is either a constant, a scalar, or a unary negation expression.

Recall that unary negation expressions are often created during algebraic simplification because of constant and expression canonicalization. They are then propagated during copy propagation so that they can combine with additive operators in the new context during further algebraic simplification.

4.5 Basic Block Scheduling

HiLO's basic block scheduler applies one of two algorithms to every function and loop body. First, it can use the list scheduling algorithm with a reservation table as described in [6]. In order to use this algorithm, the input program must be translated to three-address form and instruction latencies and other hardware parameters must be assumed. Table 1 lists these simulated hardware parameters and their default settings. All of them can be overridden on the command line. So, in the absence of software capable of automatically detecting appropriate values for these parameters, a search can easily be performed over them. The latter is our chosen experimental method, as discussed in section 5.

Table 1. Simulated hardware parameters and their default settings

Parameter	Default
integer addition latency	1
integer multiplication latency	2
floating point addition latency	2
floating point multiplication latency	4
load latency	6
store latency	7
integer ALUs	2
floating point units	2
load issue slots	8
store issue slots	8

The second algorithm simply tries to put uses as close to definitions as possible. This alternative was inspired by [7], which shows good performance with a scheduler that provably minimizes register spills no matter how many registers the target architecture has, provided that the size of the FFT transform is a power of 2. That algorithm is not implemented here; we investigated another algorithm that still tries to take other simulated hardware parameters into account, as described below.

First, the dependence DAG for the block is constructed, and the roots of the DAG are added to a queue in decreasing order of their delay to the end of the block (as would be calculated by the list scheduling algorithm, traversing the DAG to the leaves and adding instruction latencies along the way). The instructions in the queue are those instructions that need not wait for any other instructions in the block to be scheduled before they can be correctly scheduled. Next, the first instruction is taken from the queue and scheduled. In keeping with the reservation table, this instruction now occupies a particular hardware component for a particular latency, as determined by the type of operation being performed. Counters associated with every DAG child of every instruction that has already been scheduled are then incremented. If the most recently scheduled instruction has any children that can now be correctly scheduled (because all of their parents in the dependence DAG have already been scheduled), those children are then added to the queue such that the queue remains sorted by decreasing counter value from now on.

From this point further, the next instruction to be scheduled will always be the instruction in the queue with highest counter value that can also be scheduled legally given the simulated hardware's availability. The process of selecting an instruction for scheduling, incrementing counters, and adding newly schedulable children from the dependence DAG to the queue is repeated until all instructions are scheduled.

4.6 Software Pipelining

The software pipelining algorithm implemented in HiLO is called circular scheduling [8], and it is applied only to the bodies of the innermost loops. In circular scheduling, we reason that the instructions in a given loop can be viewed as a circular list, and that instructions from the top of the list (the roots of the dependence DAG, more precisely) can be circled to the bottom of the list, where they represent themselves one iteration later. Instructions moved in this way are also added to the prolog of the software pipeline, and all those that were not moved are added to the epilog. Now, the loop can be re-analyzed to determine if the basic block scheduler can produce a better schedule. If more improvement is desired the process can then be repeated.

In our implementation, we execute this circular code motion only once, reasoning that after loop unrolling, the basic block scheduler should already have enough instructions available to produce a good schedule. However, the number of dependence DAG roots that are circled can be specified as a percentage of the total DAG roots available on the command line. Thus, we give the user the

option to circle fewer instructions instead of more. The typical FFT code loop can have many dependence DAG roots already, and at some point, it's possible to circle too many instructions. As pointed out in [8], if we circle more and more instructions, we will eventually end up back at the same schedule we started with.

4.7 Engineering FFT Optimization

The preceding subsections described the classical optimizations we implemented along with some tweaks. Those tweaks were designed with an overall optimization strategy in mind. That strategy is based on the mutually beneficial relationships that different optimization passes can have. For instance, we have already seen the ways that algebraic simplification can bolster copy propagation and the ways that copy propagation can then create new opportunities for algebraic simplification in return.

Alternating between these two optimization passes, the code will eventually reach a form where no further changes can be made. This can be seen easily by noting that each new opportunity created by one pass for another makes the code smaller in some way. Algebraic simplification creates newly propagatable constants by folding them and scalar variables by stripping away a multiplication by one or an addition of zero. The propagation of constants and scalars may not make the code smaller, but algebraic simplification will not reverse the effect either, and it will use the effect to further reduce the size of the code. Lastly, the propagation of unary expressions leads to the elimination of extraneous additive operators. Since the code can only get smaller, the process of alternating between the two passes must eventually terminate.

Common subexpression elimination and copy propagation enjoy the same mutually beneficial relationship. For example, let's say two additions $b + c$ and $y + z$ are calculated, and then both the addition expressions as well as the variables their results were stored in appear later in the code, as in the upper left hand box in Figure 1. A single CSE pass will extract the addition expressions, resulting in the upper right hand box of Figure 1. Note that the variables a and x now have copies that are ready to be propagated. After a copy propagation pass, it is clear that the code in the lower left hand box can benefit from another CSE pass, which can make more variables ready for copy propagation.

In both optimization iteration scenarios described above, the code will eventually reach a form where no further progress can be made. Furthermore, there is no finite number of iterations of such "optimization loops" for which all codes will reach their fully reduced form. An arbitrary code can require an arbitrary (but finite) number of iterations of each loop, but that number can be bounded with regard to the optimization loops described here.

In the case of algebraic simplification and copy propagation, we can only say that the number of iterations applied to a given basic block in three address form is bounded by the number of instructions in that block. Now consider CSE and copy propagation applied in a loop to a basic block in three address form. What is the requirement for a piece of code to necessitate more than one iteration

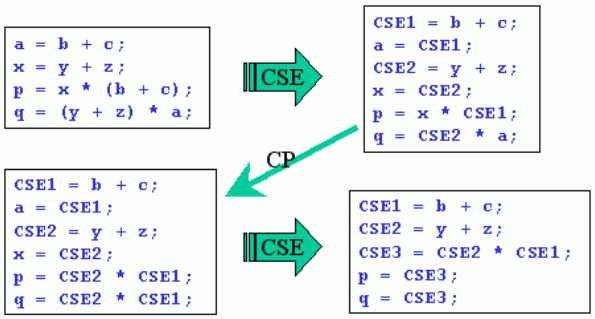

Fig. 1. The mutually beneficial relationship of CSE and copy propagation

of the optimization loop? At a minimum, one expression must be extracted by CSE so that two new copies are ready for propagation. Those copies can then combine alternately in two later instructions that involve a common second argument, and another CSE pass will be necessitated for those two instructions. In fact, every successive pair of instructions can involve one common argument and one argument that represents a copy of a common expression discovered by the previous iteration. Therefore, the number of iterations of this optimization loop applied to a basic block in three address form is no greater than $\lfloor \frac{i}{2} \rfloor$, where i is the number of instructions in that block.

5 Experiments

FFT codes are at their fastest when all their loops are fully unrolled. However, for larger formulas, the size of the resulting code can become prohibitive, so loops are preserved. Hence, the following experiments pit HiLO against SPL in both the straight-line and loop code settings. The results will then be indicative of HiLO's ability to simplify when only simplification makes a difference, as well has HiLO's scalability to large formulas when loop based optimizations are also important.

Table 2 describes the platforms used in our experiments. These platforms will hereafter be referred to by their ISA names. A collection of 45 FFT algorithms were selected to test HiLO's performance against SPL's. For each formula in each experiment, the `C` codes produced by HiLO and SPL were each sent the same back-end compiler with the same command line arguments, whatever was appropriate for the target platform as listed in table 2. The execution time of the resulting program was then plotted as a function of the algorithm number. All execution times were averaged over 10,000 trials.

5.1 Straight-Line Code

To test HiLO's performance on straight line code, SPL was set to generate both optimized and unoptimized versions of the 45 FFT algorithms of size 32

Table 2. Experimental platforms, compilers, and compiler arguments

ISA	SPARC	MIPS	x86
CPU	UltraSparcIII	MIPS R12000	Pentium IV
Clock speed	750 Mhz	300 Mhz	3 Ghz
OS	Solaris 7	IRIX64 6.5	Linux kernel 2.4.21-15EL
Compiler	Forte Developer 7	MIPSpro 7.3.1.1m	gcc 3.2.3
Compiler arguments	`-fast -xO5`	`-O3`	`-O3`

after fully unrolling them. Then, HiLO performed its optimizations on the unoptimized versions, and we then compare the resulting performance against the performance of the optimized versions. This experiment was repeated on the three target architectures. In each experiment, HiLO was invoked with the following optimization schedule: (1) array scalarization, (2) register renaming, (3) "optimization loop" over algebraic simplification and copy propagation, (4) "optimization loop" over common subexpression elimination and copy propagation, (5) dead code elimination, (6) algebraic simplification.

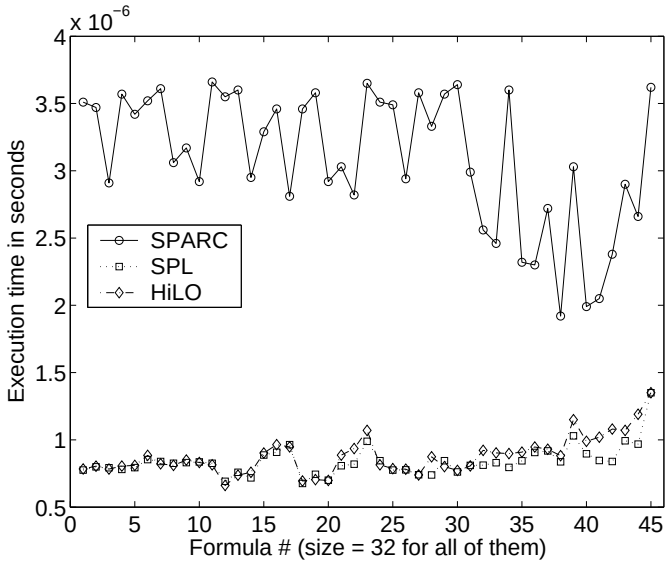

Fig. 2. Straight-line FFT performance on SPARC

In figures 2, 3, and 4, the execution times for each of the 45 selected formulas are depicted as compiled by the native compiler alone, SPL and then the native

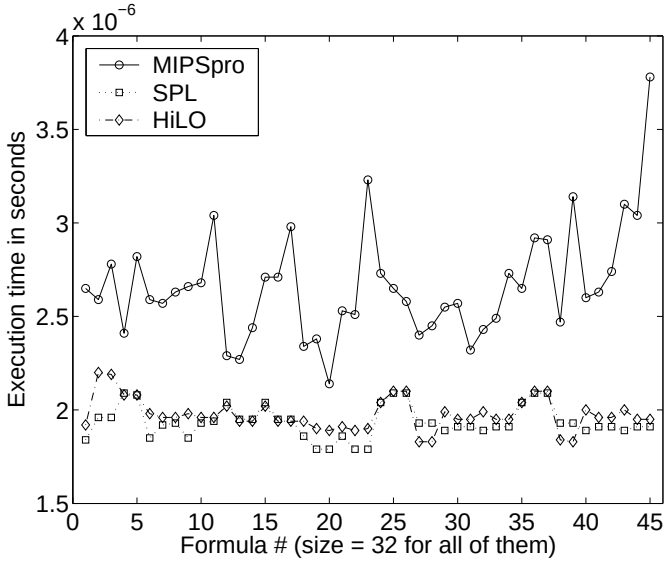

Fig. 3. Straight-line FFT performance on MIPS

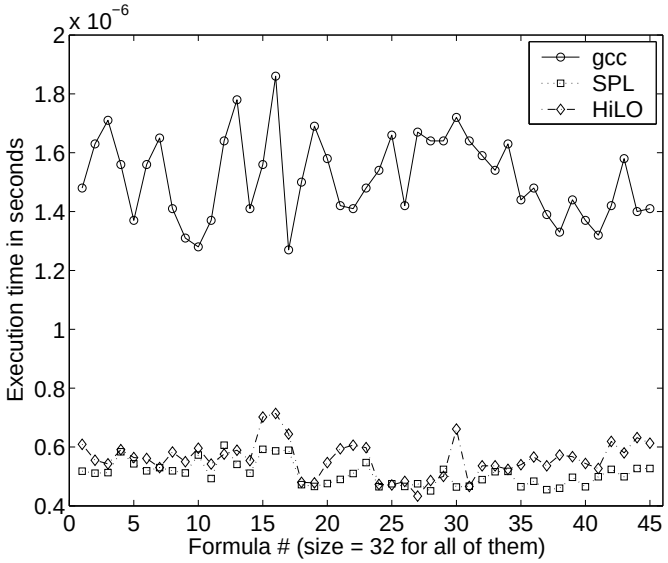

Fig. 4. Straight-line FFT performance on x86

compiler, and finally HiLO and then the native compiler. The best execution times achieved by HiLO on SPARC, MIPS, and x86 were 6.61×10^{-7}, 1.83×10^{-6}, and 4.33×10^{-7} seconds respectively. SPL's best times were 6.76×10^{-7}, 1.79×10^{-6}, and 4.51×10^{-7} seconds respectively. We also experimented with FFTW 3.0.1 using its exhaustive search option. FFTW's times were 8.66×10^{-7}, 1.91×10^{-6}, and 4.28×10^{-7} seconds respectively. All three systems achieve significant performance increases and HiLO is competitive with or better than both SPL and FFTW on all three platforms.

5.2 Loop Code

To test HiLO's performance on codes with loops, SPL was set to generate both optimized and unoptimized versions of the 45 FFT algorithms *without* first fully unrolling their loops. Of course, doing *some* unrolling is the only way to get good results out of software pipelining. So, when generating the optimized versions, SPL was allowed to unroll the innermost loops in each algorithm to the same degree that HiLO did before doing its software pipelining.

HiLO then compiled the algorithms with the following optimization schedule: (1) array scalarization, (2) register renaming, (3) "optimization loop" over algebraic simplification and copy propagation, (4) dead code elimination, (5) induction variable elimination, (6) "optimization loop" over CSE and copy propagation, (7) dead code elimination, (8) loop unrolling with factor 4, (9) "optimization loop" over algebraic simplification and copy propagation, (10) array scalarization, (11) register renaming, (12) copy propagation, (13) dead code elimination, (14) software pipelining, (15) copy propagation.

Table 3. Simulated hardware parameter settings used in HiLO's search

Parameter	**Settings included in search**				
integer ALUs	1	2			
floating point units	1	2			
CSE preserves subscripts	no	yes			
scheduling algorithm	list	HiLO's own			
pipelining	0.25	0.5	0.75	1.0	just scheduling

Then, for each of the 45 algorithms, HiLO searched over several of the simulated hardware parameters, the CSE subscript preservation option (see section 4.3), and the software pipelining parameter (see section 4.6) by repeating the above optimization schedule with different settings and re-measuring the resulting performance. Table 3 lists some of the parameters that are searched over and the values those parameters are allowed to take. For every possible combination of parameter settings from table 3, we tried setting the latencies to their default values and setting them all equal to 1. In addition, we included in

the search an optimization schedule identical to the one given above but with software pipelining removed entirely.

This entire search process was then repeated on all three target architectures. The results are depicted in figures 5, 6, and 7, and they show HiLO gains a clear performance advantage when performing software pipelining on both the SPARC and MIPS architectures. On the x86 architecture, HiLO remains competitive. When viewing these results, it should be kept in mind that the initial loop unrollings performed by SPL are very similar, but not identical to those performed by HiLO. In some instances, it may be the case that an extra unrolled loop in HiLO accounts for extra performance benefits when compared with SPL.

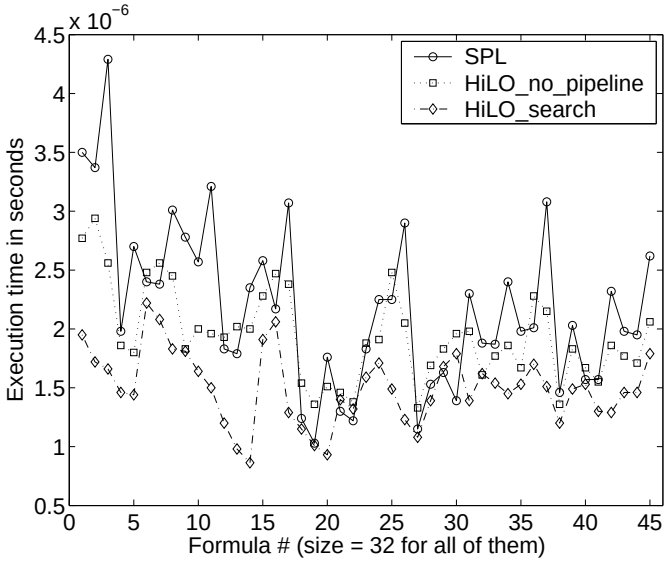

Fig. 5. Pipelined FFT performance after parameter search on SPARC

Solid conclusions about software pipelining can be reached by comparing the two HiLO generated data sets on these graphs. The data labeled "HiLO no pipeline" depict HiLO's performance when using the afore mentioned optimization schedule with the pipelining pass removed. Therefore, it is pipelining with some simulated hardware parameter settings that accounts for the significant improvements seen in almost every formula on both SPARC and MIPS when compared against the non-pipelining HiLO optimization strategy. And the actual values used by different formulas for the parameters in table 3 to achieve the improved performance are quite varied from formula to formula, except for CSE's subscript preserving parameter, which stayed on consistently.

It is also interesting to note that x86 did not respond well to software pipelining. The search process almost always settled on a configuration that involved

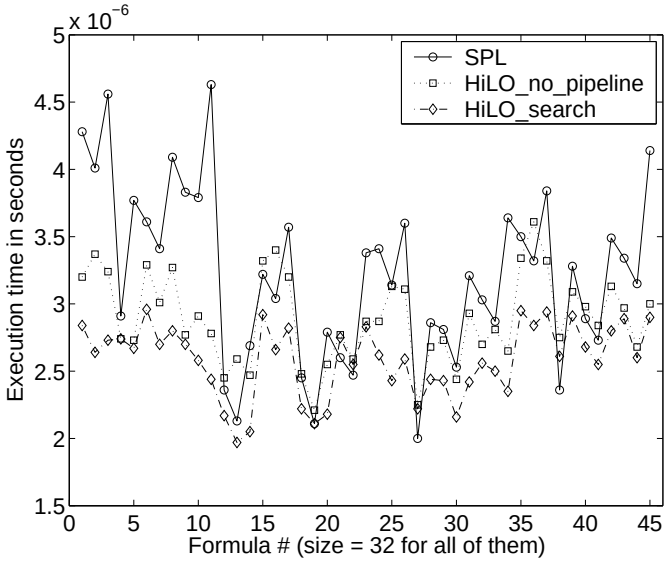

Fig. 6. Pipelined FFT performance after parameter search on MIPS

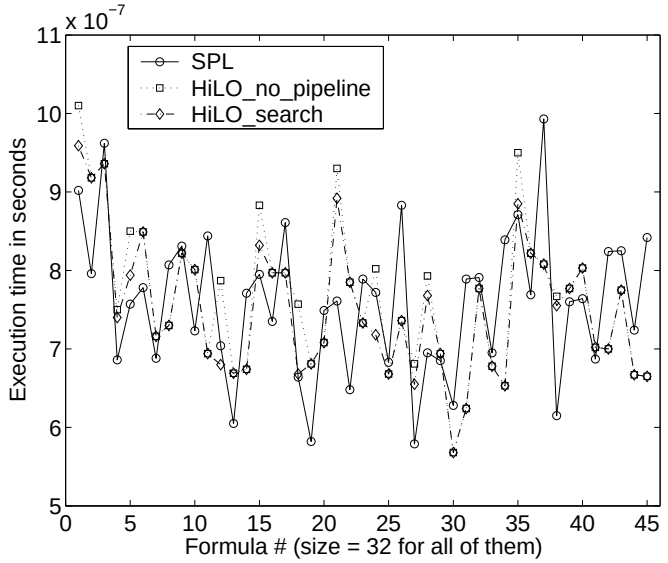

Fig. 7. Pipelined FFT performance after parameter search on x86

no pipelining or scheduling on this architecture. Together, all of these results lead us to believe that imprecision of dependence analysis and the difficulty of finding good instruction schedules lie at the heart of native compilers' struggles with FFT codes.

6 Conclusion

We have presented the HiLO high level optimizer and shown that it is a good framework for researching and applying the effects of high level optimizations on domain specific codes. Following in the tradition of domain specific compilers such as SPL and FFTW, we have shown originally that native compilers can be coaxed to produce even more efficient code through software pipelining. Finally, we believe the trends discovered in our experience with searching for a good software pipelining are further evidence that arrays, dependence analysis, and instruction scheduling are the keys high performance in FFT.

In the near future, we plan to expand the domain on which HiLO is applicable to other DSP transformation algorithms. If our techniques prove effective on these related codes, they can be integrated into the SPIRAL automatic tuning system where they will be of greater value.

References

1. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. In: to appear in Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation". (2005)
2. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. Volume 3. (1998) 1381–1384
3. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing **27** (2001) 3–35
4. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: A Language and Compiler for DSP Algorithms. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (2001) 298–308
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc. (1995)
6. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Inc (1997)
7. Frigo, M.: A Fast Fourier Transform Compiler. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (1999) 169–180
8. Jain, S.: Circular Scheduling: A New Technique to Perform Software Pipelining. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (1991) 219–228