THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

Interprocedural Optimizations for Embedded Systems

Yuhong Wang

A project submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Masters of Science

April 8, 1999

ACKNOWLEDGMENTS

First and foremost, I want to thank Dr. David Whalley for taking me as his student early in my graduate study in this department. This gave me a sense of belonging and direction more than anything else. Because of him, I was able to make better use of time that would have been wasted on computer games otherwise. Then in summer of 1998, he gave me the opportunity of a lifetime to work with the compiler group at Lucent Technologies. I was writing parts of a compiler before I even knew what a compiler was and using a programming language I never had the chance to learn before: C. Fortunately, everything worked out for the best. Before long, *printf* started to look friendlier than *cout* to me. Always trying to provide me with the best, Dr. Whalley got me on a research assistantship funded by Lucent Technologies in the fall of 1998. Ever since, I have been able to concentrate on my master's project. If I had to teach while working on the project, it would have been far from completion at this point.

I also want to thank Dr. Xin Yuan and Dr. Robert van Engelen for teaching me valuable knowledge as well as serving on my committee. Even though I had been exposed to compiler writing before I took the compilers course with Dr. Engelen, he taught me the theoretical foundations of compilation and gave me the motivation be a good compiler writer.

I would also like to take this opportunity to thank my husband for encouraging me to change my major to computer science, which has turned out to be something that I really enjoy and something in which I do better than anything else that I have touched upon.

Last but not least, I am thankful to my parents for the effort and *dollars* they spent to send me abroad for a better education. I am especially grateful to them for allowing me to come to the other side of the Earth considering they did not even let me go to college out-of-town. Although this day did come very soon, now I can finally make them proud.

Contents

1 Introduction

The mission of this project is to improve assembly code for the Digital Signal Processor 16000 (DSP16k). DSP16k processors are widely used in embedded systems such as cellular phones and modems. One important characteristic of such systems is their limited amount of memory. Consequently, optimizations that decrease execution time but increase code size substantially should be applied with caution.

DSP applications are usually compiled and loaded into the devices once and for all. The implication is that the speed of the compiler is of less concern than the speed of the code produced. Therefore, we can afford algorithms with high complexities to perform optimizations.

The compiler used for the DSP16k is called *cc16k*, which is a C compiler retargeted to the DSP16k processor. The *cc16k* compiler already performs some basic optimizations, e.g., function inlining. However, since the source files are compiled individually, at the time of the compilation of each source file, the compiler does not have information about the functions that are defined in other source files. Thus, many optimizations cannot be fully exploited. For example, inlining is performed by the *cc16k* compiler within the same source file but the *cc16k* compiler cannot delete the functions inlined in case they are called from functions defined in other source files.

Figure 1 shows the approach used to address this problem in our optimizing compiler. First, all the source files of a DSP16k application are still individually compiled with the *cc16k* compiler to obtain an assembly file for each source file. Next, the optimizer reads in all these assembly files and processes them as a whole. This way, the optimizer has a global picture of all the functions that comprise the application. The output from the optimizer is a single assembly file with improved code. The analysis and optimizations performed by the optimizer that are relevant are listed in Figure 2.

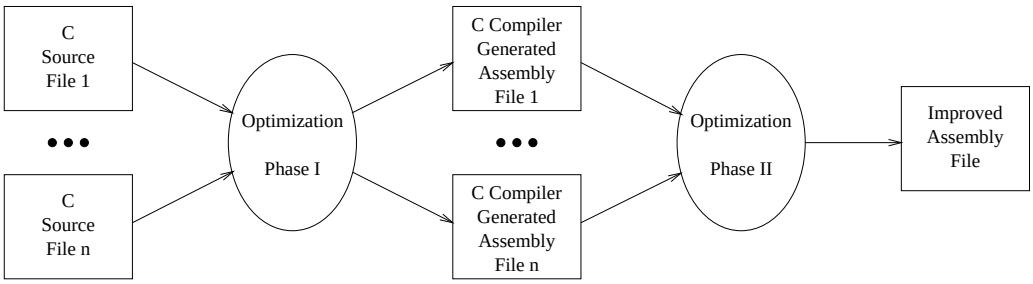

Figure 1: Overview of the Optimizing Compiler

Because of the fact that the optimizer knows all the functions defined in an application, it can afford to perform optimizations that require interprocedural analysis. In particular, four such optimizations are described in this document: removing dead code, inlining functions, eliminating unnecessary saves and restores of registers, and converting far transfers of control to near transfers of control. These optimizations are highlighted in Figure 2. The effectiveness of these optimizations is measured both in percentage of code size saved and in percentage of execution cycles saved.

2 Removing Dead Code

In this report *dead code* refers to any code portion that is never executed. One of the reasons that dead code exists in *cc16k*-generated code has to do with the way that the *cc16k* compiler inlines functions. As mentioned in section 1, the *cc16k* compiler compiles one source file at a time and

functions are only inlined within the same source file. When the *cc16k* compiler inlines a function, it copies code from the callee into the caller, and keeps the function body of the callee in case it is called from other files. If the function is only called from one place, it becomes dead code.

2.1 The Main Algorithm

Two steps are taken in the process of dead code elimination. In the first step, all basic blocks in an application that are reachable from the first basic block of the main function during the execution are marked as such. In the second step, all basic blocks that are not marked are deleted. Before we get into how the basic blocks are marked and deleted, it is worth explaining what we mean by saying that one block is *reachable* from another. First of all, there are three ways that block A can have a *transition* to block B:

1. B is the fall-through block of A.
2. B is the target of a branch or jump at the end of A.
3. B is the top block of the function called from A.

If either of the first two cases is true, block B is called a *successor* of block A. Now we can say that block B is *reachable* from A if:

1. There exists a transition from A to B, or
2. X is reachable from A and there exists a transition from X to B.

Figure 3 lists the algorithm used for dead code removal. All C programs start their execution from the main function. At the assembly code level, the execution starts from the first basic block of the main function. We refer to this block as block M in this discussion. If we use a node to represent each basic block and an edge from node A to node B to represent the fact that there is a transition from A to B, then all the blocks that are reachable from block M form a directed graph rooted at M. On this graph, the children of a node represent the blocks that can be reached in a single transition from it in the three ways mentioned above. The reachable nodes from M can be marked in a depth-first traversal of the graph. Whenever we visit a node, we mark it as reachable, and then in a recursive step we visit all its children. It is critical that a node be marked before any

```
PROCEDURE main
    read in assembly files
    removedeadcode
    build call graph
    find loops
    determine loop iterations
    inlineforcaching
    cache loops
    pushupregisters
    inlinesinglecallsite
    fartonear
    output a single assembly file
END PROCEDURE
```

Figure 2: Pseudocode for the Optimizer

```
PROCEDURE removedeadcode
    LET B be the list of basic blocks of an application
    FOREACH basic block b in B
        clear b as not marked
    mark (the first basic block of the main function)
    FOREACH basic block b in B
        IF (b is not marked AND b starts a function)
            put b on list L
    FOREACH basic block b in the list L
        LET f be the function to which b belongs
        IF (address of f is taken)
            mark (f)
    FOREACH basic block b in B
        IF (b is not marked)
            delete block b from B
END PROCEDURE

PROCEDURE mark (b)
    IF (b is the top block of a library function)
        RETURN
    IF (b is already marked reachable)
        RETURN
    mark b as reachable
    FOREACH successor block s of b
        mark (s)
    IF (b calls function f)
        LET t be the top block of f
        mark (t)
END PROCEDURE
```

Figure 3: Pseudocode for Dead Code Removal

of its children is visited. Otherwise, the traversal routine will end in an infinite recursion if there are cycles in the graph.

There are two base cases to the recursive algorithm above:

1. The node reached is the first block of a library function.
2. The node is visited already.

In the first base case, we stop going any further when we reach a library function, since the assembly code of the library routines is not available to us. In other words, no optimization is extended to the library routines. The second base case occurs when a basic block can be reached from more than one basic block. One simple example is that a basic block is a common successor of two different basic blocks.

2.2 Handling Indirect Calls

One complication of the marking step comes from indirect calls, i.e., function calls through pointers. This presents a problem because the address of a function can be taken and passed to a library routine and then this function can be called from the library routine. Since we do not have the assembly code for the library routines, the optimizer cannot detect these calls. Consequently, some of the reachable blocks can be missed. We have made the decision not to delete any function whose address is ever taken and thus might be called indirectly from library routines. For this reason, the marking is actually done in two passes. In the first pass, the basic blocks are marked as described above. Next, every unmarked basic block that is the top block of a function is put on a list. In the second pass, each of the function that is on the list is checked to see if its address is ever taken. If it is, a recursive marking is started from its first basic block. This will guarantee that all blocks that are reachable from the first block of this function will not be deleted.

2.3 Removing Unmarked Basic Blocks

The next step is to go through the list of all basic blocks, and delete the ones that are not marked. One thing that has to be mentioned here is that in the implementation, dead code removal is performed before other optimizations and analysis. This way, even when a complete function is removed, no other analysis results need to be updated.

One issue that has come up again and again in the implementation of the various optimizations is how to traverse a linked list while nodes are being removed at the same time. It can become complicated when multiple nodes are removed at once at different locations in the same list. The problem is that the position of the next node is stored in the current node, but the current node could be deleted. In the case of dead code removal, the problem is relatively simple: all that needs to be done is to record the position of the next node before the current node is deleted.

The last point to make about dead code removal is that even though the basic unit of code removal is a basic block, it could have been chosen to be a function. The advantage of using a basic block as the unit is that code removal is more fine-grained. With this approach, dead code within a function can also be eliminated. The disadvantage is that if dead code within a function is infrequent, the optimizer will be less efficient compared with having a function as the unit.

Figure 4 shows an example of dead code removal from the test program *edn*. The main function in the application calls a number of functions in turn. One of these functions, namely, *vec_mpy* is chosen for the purpose of illustration. Before dead code removal is applied, the assembly code of two functions generated by the *cc16k* compiler are shown. We can see that code from function *vec_mpy* has already been inlined into the main function. However, the original function of *vec_mpy* is kept by the *cc16k* compiler. After applying dead code removal, function *vec_mpy* is removed since it is not called from anywhere, and its address it never taken.

C Source Code	DSP16k Assembly Code	
	Before	After

void vec_mpy(short y[],
short x[],
short scale){
int i;
for(i=0; i<150; i++)
y[i]+=(scale*x[i])>>15;
}
 	
main(){
...
short c = 0x3;
...
vec_mpy(a, b, c);
...
}
	_vec_mpy:
	y = a0
	a2 = 0xffffff6b
	_.L11:
	xl = *r1++
	x = xl
	p0 = xh*yh p1 = xl*yl
	a1 = p1
	a1 = a1 >> 15
	a0h = *r0
	a0 = a0 >> 16
	a0 = a0 + a1
	*r0++ = a01
	a2 = a2 + 1
	if le goto _.L11
	if true return
	
	_main:
	...
	y = 0x00000003
	a2 = 0xffffff6b
	_.L25:
	xl = *r4++
	x = xl
	p0 = xh*yh p1 = xl*yl
	a1 = p1
	a1 = a1 >> 15
	a0h = *r5
	a0 = a0 >> 16
	a0 = a0 + a1
	*r5++ = a0l
	a2 = a2 + 1
	if le goto _.L25
	...
	if true return
		_main:
		...
		y = 0x00000003
		a2 = 0xffffff6b
		_.L25:
		xl = *r4++
		x = xl
		p0 = xh*yh p1 = xl*yl
		a1 = p1
		a1 = a1 >> 15
		a0h = *r5
		a0 = a0 >> 16
		a0 = a0 + a1
		*r5++ = a0l
		a2 = a2 + 1
		if le goto _.L25
		...
		if true return

Figure 4: Example of Dead Code Removal from Test Program *edn*

3 Inlining Functions

There are many benefits from inlining a function. An intuitive one comes from the elimination of the call instruction in the caller and the return instruction in the callee. If the callee is not a leaf function, another benefit comes from the elimination of the save and restore of its return address. This is possible because after the code from the callee is inlined into the caller, it is no longer a function of its own. In addition, because the caller is not a leaf function to begin with, its return address is already saved. Therefore, no additional work needs to be done even if the inlining of the callee introduces new function calls to the caller. Additional reduction of code size may be possible if the caller becomes a leaf function after all the functions that it calls are inlined, at which point the save and restore of the return address of the caller can be eliminated as well. Furthermore, inlining may provide opportunities for additional compiler optimizations. The kind of inlining discussed in Section 3.1 is done precisely for this reason.

As mentioned before, the *cc16k* compiler already does some limited function inlining. The goal of this part of the project is to be able to inline any function when it is desirable to do so. The next two subsections each discuss one situation where inlining is desirable.

3.1 Inlining Functions to Make Loops Cacheable

One situation where inlining is desirable is related to the zero overhead loop buffer (ZOLB) of the DSP16k architecture. This buffer can be seen as a cache that can contain up to 31 instructions. The instructions in the ZOLB can be executed a specified number of times without any loop overhead. Hence, placing a loop in the ZOLB has significant benefit in execution time. Unfortunately, a loop cannot be placed in the ZOLB unless it satisfies the following three rather stringent conditions:

1. It has no transfers of control except at the exit block.
2. It has a known number of iterations.
3. It has no more than 31 instructions.

A loop is said to be *cacheable* if it satisfies the above conditions and therefore can be placed in the ZOLB.

The requirement that a cacheable loop does not contain any transfer of control (TOC) instructions implies that if a loop contains a call instruction, it cannot be placed in the ZOLB. The hope is that if we inline the function called, the call instruction will be eliminated and the loop might become cacheable. On the other hand, if there are branches or jumps not in the exit block, inlining will not help. When a function is inlined to make loops cacheable, the code size may increase if the function is called from more than one place. Because caching a loop has significant performance gain, the code size increase is considered an acceptable sacrifice.

The number of iterations of a loop is a property of the loop that cannot be changed by inlining functions. Before inlining is carried out, an analysis is performed to get the number of iterations of each loop in the application. This analysis was implemented by Dr. Whalley. The number of iterations of a loop is stored in a field associated with the data structure used to represent loops.

To check the third condition, we can pretend that we are trying to inline all the functions called from the loop. If there is no reason that prevents any of these functions from being inlined, we then count the number of resulting instructions in the loop. The following modifications to the code are taken into consideration:

1. The call instruction in the caller will be removed after inlining.
2. The return instruction in the callee will be removed after inlining.

3. After the loop is cached, the branch or jump instruction that controls the loop will be removed as well. The loop will be controlled by an implicit register instead.

If all function calls in a loop can be inlined, and the resulting loop consists of no more than 31 instructions, the actual inlining is performed.

A functions cannot be inlined unless it satisfies the following requirements:

1. The function is not a library function.
2. The call is not an indirect call.
3. The function consists of a single basic block.

The first requirement is obvious since we do not have the code of library functions. The second requirement is for simplification. As long as we are not removing the body of the callee after inlining it, there is no danger. An example of an indirect call instruction is *"if true call pt0"*, where *pt0* is a register. Since the name of the function does not appear in the call instruction, it is a bit complicated to locate the body of the callee. This is due to the fact that once the address of a function is taken, it can be manipulated as data. For example, it can be stored in data structures and passed in registers between function calls. Keeping track of this address requires a difficult analysis. Considering that indirect calls are relatively uncommon in DSP applications, it was not worth attempting to inline them. The third requirement has some justification but can be relaxed as well. As mentioned before, a cacheable loop does not contain any transfers of control. By choosing to inline single-block functions only, we guarantee that no additional transfers of control are introduced into the loop. This choice also simplifies the algorithm in that no recursive inlining is necessary, since a single-block function cannot contain any functions calls. It is imaginable that we can have a loop L that has a call to function A, which has multiple blocks due to a call to function B, and that after inlining B into A, A becomes a single block function, which can in turn be inlined into loop L. However, a function can be of multiple blocks because of branches or jumps as well, which will prevent a loop from being cached. Furthermore, a cacheable loop contains no more than 31 instructions. Even though recursive inlining may result in a loop with no transfers of control, it is likely to result in more instructions than a ZOLB can hold. For these reasons, I chose not to inline multiple-block functions. In the next section, we will be inlining all functions called from a single site. Without the constraints that are imposed by the ZOLB, we will be able to do a much more thorough job in inlining functions. For example, recursive inlining and inlining of multiple-block functions are all performed.

Figure 5 gives the algorithm for inlining functions to make loops cacheable. The algorithm summarizes the important points discussed above. When we actually inline a function, the code from the callee is simply copied to the place where it is called. Afterwards, if the address of the callee is never taken and the number of call sites of the callee becomes zero after inlining, we delete the inlined function.

Figure 6 gives an example of a DSP16k assembly program that consists of two functions: *_abs* and *_main*. Note that only part of *_main* is shown. Function *_main* has a loop that starts at the label *_L5* and ends in the branch instruction "*if le goto L5*". This loop has two blocks, the first block ends in a call instruction, and the rest of the loop comprises the second block, which is the exit block of the loop. Because the call instruction is a TOC instruction that does not belong to the exit block, the loop cannot be placed in the ZOLB. If we inline function *_abs*, the call instruction will be replaced with the body of function *_abs* not including the return instruction. After inlining, the loop will satisfy all the conditions for it to be cached. Figure 6 shows the program before and after function *_abs* is inlined. Notice that after inlining, function *_abs* is deleted because it is not called from elsewhere. The cached version of the loop is not shown here, since caching a loop is not part of this project. But it was done by Dr. Whalley as a separate optimization in the optimizer.

```
PROCEDURE inlineforcaching
    FOREACH loop L in the application
        IF (need_inline (L))
            FOREACH call instruction in loop L
                inline the function called
    FOREACH function f in the application
        IF (f.callsites == 0 AND address of f is not taken)
            delete f
END PROCEDURE

PROCEDURE need_inline (L)
    IF (L does not have a known number of iterations)
        return FALSE
    FOREACH basic block b in loop L
        IF (b ends in a branch OR b ends in a jump)
            return FALSE
        IF (b has a call)
            IF (the function called is a library function)
                return FALSE
            IF (the call is indirect)
                return FALSE
            IF (the function called is multi-block)
                return FALSE
            hasdirectuserfunccalls = TRUE
    IF (hasdirectuserfunccalls == FALSE)
        return FALSE
    numinsts = 0
    FOREACH basic block b in loop L
        FOREACH assembly line m in b
            IF (m is not a call)
                numinsts ++
            ELSE
                FOREACH assembly line n in the function called
                    IF (n is not a return)
                        numinsts ++
    numinsts --   # the branch instruction will be removed after caching
    IF (numinsts <= 31)
        return TRUE
    ELSE
        return FALSE
END PROCEDURE
```

Figure 5: Pseudocode for Inlining to Make Loops Cacheable

However, it is worth mentioning again that after caching, the branch instruction "*if le goto L5*" will be removed. This is important because it affects our calculation that determines whether a loop is cacheable.

Before		After	
_abs:	**a0 = a0**		
	if lt a0 = -a0		
	if true return		
_main:		_main:	
	...		...
	r4 = _a		r4 = _a
	a5 = 0		a5 = 0
	a4 = -9999		a4 = -9999
L5:	a0 = *r4++	L5:	a0 = *r4++
	call _abs		**a0 = a0**
	a5 = a5 + a0		**if lt a0 = -a0**
	a4 = a4 + 1		a5 = a5 + a0
	if le goto L5		a4 = a4 + 1
	...		if le goto L5
	if true return		...
			if true return

Figure 6: Example of Inlining to Make Loops Cacheable

3.2 Inlining Functions Called from A Single Site

To inline functions called from only one site, we first need to determine at how many sites each function is called. A function can either be called explicitly, i.e., by its name, or it can be called implicitly, i.e., through a pointer. Explicit calls are relatively easy to count, since each call contains the name of the function called. Implicit calls, on the other hand, require some thought. If a function is called through a pointer, then the address of the function is taken and assigned to a register. This register can be passed to a library function as an argument, in which case, the function can be called by the library function indirectly. Without the code of library functions, it is impossible to determine the number of sites where a function is called indirectly. Furthermore, as is explained in Section 3.1, even if the address of a function is not passed to any library function, it can still be difficult to determine which function is invoked at each indirect call. For these reasons, we do not inline any function whose address is ever taken. This way, we will not accidentally delete a function that is only called indirectly from library routines. To summarize, the following lists the criteria we use to determine if a function can be inlined for the purpose discussed in this section:

1. The function is not a library function.
2. The address of the function is never taken.
3. The function is only called from one site.

3.2.1 The Basic Algorithm

The basic algorithm for in-lining functions that are called from a single site is given in Figure 7. This algorithm is based on the following observation. If we use a node to represent each function,

and an edge from A to B to represent a call from function A to function B, then the functions and calls in an application form a directed graph, which is referred to as the *call graph*. The root of the call graph is the main function. To inline all functions that satisfy the criteria listed above, the nodes on the call graph are traversed in a depth-first fashion. When a node is being visited, all its children that can be inlined are inlined immediately. If a child cannot be inlined, a recursive call is made on the child.

```
PROCEDURE inlinesinglecallsite (parent)
    IF (parent is a library function)
        RETURN
    IF (parent is already visited)
        RETURN
    mark parent as visited
    FOREACH child c called by parent
        IF (c is not a library function AND
            the address of c is not taken AND
            c.callsites == 1)
            mergecode (parent, c)
            # after mergecode(), functions called from c are added to the end
            # of parent's list of children
        ELSE IF (parent and child are not the same function)
            inlinesinglecallsite (c)
END PROCEDURE

PROCEDURE setcallsites
    FOREACH function f in the application
        f.callsites = 0
    FOREACH function f in the application
        FOREACH function p called by f explicitly
            IF (p is not a library routine)
                p.callsites ++;
END PROCEDURE
```

Figure 7: Pseudocode for Inlining Functions Called from A Single Site

It is worth pointing out that when a child is inlined into its parent, all the functions called by the child become the functions called by the parent. Therefore, not only is the call graph changed because of the removal of the node for the child, it is also changed due to the fact that all the nodes that are called by the child move one level up and become the children of the parent. If we look closely at the implementation, we notice that the main algorithm is recursive, and that within each recursion, there is a loop that guarantees that all the children of the parent being visited will be considered for inlining. This loop goes through a list of the functions that are called explicitly by the parent. If we inline a child that have some function calls of its own, these functions will become functions called by the parent, and therefore should be added to the list of the calls made by the parent. These functions cannot be inserted at arbitrary positions of the list. For example, if they are added at the beginning of the list, since the loop has already passed that point, these newly added children of the parent will not be considered for inlining. For this reason, they are appended at the end of the list to guarantee that the loop will eventually come to examine them.

The example given in Figure 8 will make the execution of this algorithm clear. Six snapshots

of the call graph of an application during the inlining process are shown. For clarity, only seven functions that are relevant are chosen. Snapshot (1) shows the call graph of seven functions before the algorithm is applied. Node M represents the main function, and the asterisk next to function A indicates that the address of A is taken. The recursive algorithm starts at M. During the recursion on M, all its children are considered for the possibility of being inlined. Since the address of A is taken, it cannot be inlined into M. Instead, a recursive call is invoked on A, which first results in the inlining of C into A. Snapshot (2) is the call graph after C is inlined. Next, D is inlined in a similar way. As the result of the inlining of D, the functions called by D, namely, E and F, become functions called by A. This change shows up on the call graph in snapshot (3) as the adoption of E and F by A after D is removed. Now that E and F are children of A, they are considered for inlining as well. The next two snapshots, i.e., (4) and (5) show the call graph after E is inlined and after F is inlined, respectively. At this point, all the children of A have been inlined, so the recursion on A returns to M, where the other child of M, namely B, is inlined. When the algorithm stops, two functions remain. M is the main function, therefore, cannot be inlined. A remains because its address is taken.

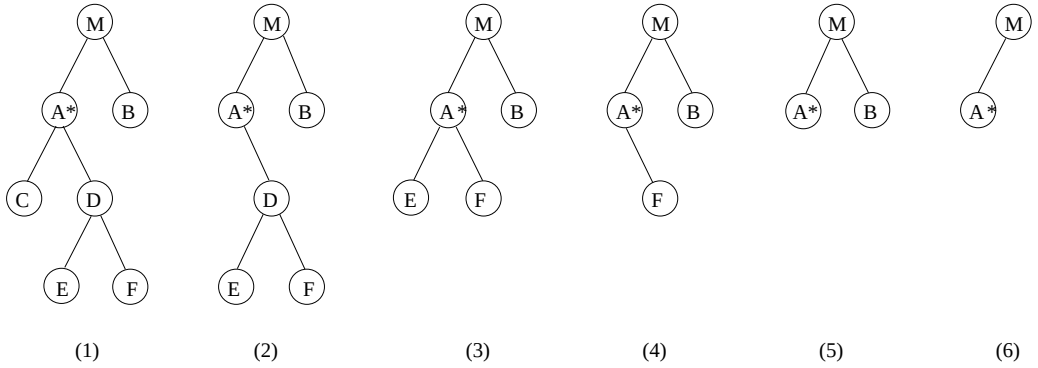

Figure 8: Example 1 of Inlining Functions Called from A Single Site

3.2.2 Recursive Functions in the Application

Another point to make about the main algorithm is that whenever we visit a parent node, before any of its children is considered for inlining, we mark the parent as visited. This is to prevent infinite recursions in the case where there are recursive functions in the assembly code being optimized. This marking will also prevent nodes that are reached from multiple parents from being examined repeatedly for the possibility of inlining.

The example given in Figure 9 focuses on functions called from more than one site and recursive functions in the application. Eight snapshots of the call graph of an application during the inlining process are shown. In each snapshot, the dark circle is the parent being visited, and the dashed circle is the child being considered for inlining. Snapshot (1) shows the moment when the recursion is on M, and A is being considered for inlining into M. Of course, A cannot be inlined because it is called from two sites, i.e., M and B. As a result, a recursive call is made on A. In snapshot (2), C is being considered for inlining into A. Because the address of C is taken, it cannot be inlined either. The recursion goes deeper onto C where E is the candidate for inlining, as illustrated in snapshot (3). E is only called from C and its address is not taken, therefore, it can indeed be inlined into C. Snapshot (4) shows the call graph after E is inlined, and the recursion retracts to M where B is the candidate for inlining. Since B is called from M and D, it is not inlined. In snapshot (5),

the recursion moves to B, and A is examined for the possibility of being inlined. Again, A is not inlined due to the fact that it is called from two sites. Now the recursion moves to A for the second time. Only this time A is already marked as visited, so the recursion returns before its child C is considered redundantly for inlining. This is illustrated in snapshot (6) by the absence of a dashed circle. When the recursion returns to B, as shown in snapshot (7), the other child of B, i.e., D is the one to be inlined. Snapshot (8) shows the call graph after D is inlined into B. There are two things that are quite interesting in snapshot (8). First, after D is inlined, the call from D to B becomes a call from B to itself, hence the arrow from B to B. Second, since B calls itself, in other words, it is its own child, it is considered for inlining just like any other children of B's. This is why the circle around B is both dark and dashed. Of course, an immediate recursive function that is reachable from the main function can never be inlined because it is called from at least two places. If it were not reachable from the main function, it would not have been reached by the algorithm.

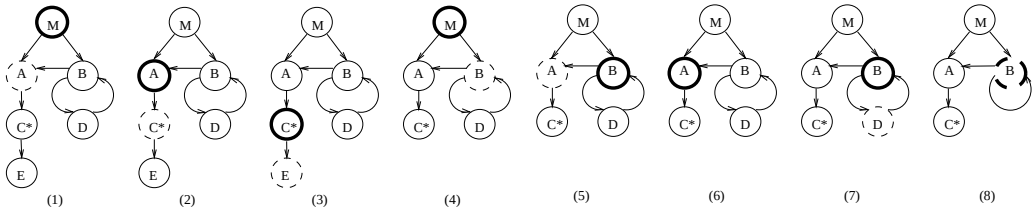

Figure 9: Example 2 of Inlining Functions Called from A Single Site

3.2.3 Updating Data Structures

The hard part of inlining functions called from only one site is to keep all the data structures associated with different levels of abstraction of the application being optimized in a consistent state. Almost every data structure internal to the optimizer is being affected by this optimization. Just to mention a few obvious ones, lines of assembly code are deleted and moved around, basic blocks are moved around and merged together, predecessor-successor relationships are changed, and functions are deleted. What really makes the matter worse is that these data structures are interwoven. Before changing one item, careful analysis has to be done as to what will be affected by this change. Without going into too much detail of the implementation, the following gives a list of things that have to be accomplished:

1. Remove the save and restore of the return address of the callee.
2. Remove the last instruction in the callee if it is an unconditional return.
3. Unhook basic blocks in the callee and insert them into the caller.
4. Update the field in the data structure for basic blocks that stores the function to which a basic block belongs for all the blocks inlined from the callee.
5. Fix predecessor-successor relations among the *call block* (the block in the caller that made the call), the basic block following the call block, the top block from the callee, and the return block from the callee.
6. Convert conditional and unconditional returns in the code from the callee to branches and jumps to the block after the call block, respectively.
7. Merge the top block of the callee with the call block of the caller.

8. Merge the call block with its next block if the callee is a single-block function that does not end in a TOC instruction and that the the next block does not have a label.

9. Merge the last block (if different from the top block) from the callee with the block after the call block if the former does not end in a TOC instruction and the latter does not have a label.

10. Remove the call instruction from the caller.

11. Decrement the number of call sites of the callee by 1.

12. Functions called explicitly by the callee become functions called explicitly by the caller.

13. Functions called implicitly by the callee become functions called implicitly by the caller.

14. Delete the save and restore of return address of the caller if it becomes a leaf function.

15. Delete the callee.

Two key points will be made about the above list of operations here. First, the *cc16k* compiler may generate multiple return instructions in a single function. These return instructions may be unconditional or conditional. For different types of return instructions, readers are referred back to Table 1. After the code of the callee is inlined into the caller, if the last basic block from the callee contains an unconditional return, it is no longer needed and therefore it is removed. Other returns still play important roles in controlling the flow of execution of the application and thus cannot be removed. To preserve the semantics of these returns after inlining, conditional returns and unconditional returns are converted into branches and jumps, respectively. The target of these branches and jumps is the block after the call block. A label may need to be added to this target block if it does not already have one. Because of these added branches and jumps, new predecessor-successor relations are introduced. Figure 10 shows an example where a function with multiple returns is inlined. After inlining, the conditional return "*if pl return*" is converted to a branch "*if pl goto _.L01*", where *_.L01* is a label newly assigned to the block after the call block.

The second point to make is that to be able to carry out the above operations correctly, it is essential that the partial orderings among some operations are preserved. The above list gives one correct ordering. For example, one cannot merge the bottom block from the callee with the block after the call block before converting returns from the callee to branches or jumps. Again, we can use Figure 10 to illustrate this point. According to the semantics of the assembly code, the conditional return should transfer control to the instruction "*a5 = a5 + a0*". If the bottom block from the callee were merged with the block after the call block first, the instructions from "*a0 = -a0*" to "*if le goto _.L5*" would belong to the same basic block and the instruction "*a5 = a5 + a0*" would appear in the middle of the basic block. If we attempted to convert the conditional return to a branch after that, we would have a transfer of control into the middle of a basic block, which would violate the definition of a basic block.

4 Optimization on Nonscratch Registers

4.1 Moving Saves and Restores to Callers

The calling convention for the DSP16k is that the caller is responsible for all scratch registers it wants saved, and the callee is responsible for all nonscratch registers. Examples of nonscratch registers include *a4*, *a5*, *a6*, *a7*, *r4*, *r5*, and *r6*. When the *cc16k* compiler generates code, it saves all nonscratch registers that are used by a function. However, this is not always necessary. For

Before		After	
_main:		_main:	
	sp−−2		sp−−2
	*sp−−2 = a4		*sp−−2 = a4
	*sp−−2 = a5		*sp−−2 = a5
	*sp−−2 = r4		*sp−−2 = r4
	*sp = pr		*sp = pr
	r4 = _a		r4 = _a
	a5 = 0		a5 = 0
	a4 = 0xfffffc19		a4 = 0xfffffc19
_.L5:		_.L5:	
	far call _abs		a0 = a0
	a5 = a5 + a0		**if pl goto _.L01**
	a4 = a4 + 1		a0 = -a0
	if le goto _.L5	**_.L01:**	
	a0 = a5		a5 = a5 + a0
	pr = *sp++2		a4 = a4 + 1
	r4 = *sp++2		if le goto _.L5
	a5 = *sp++2		a0 = a5
	a4 = *sp++2		pr = *sp++2
	if true return		r4 = *sp++2
_abs:			a5 = *sp++2
	a0 = a0		a4 = *sp++2
	if pl return		if true return
	a0 = -a0		
	if true return		

Figure 10: Example of Inlining Functions with Multiple Returns

example, if after the call a nonscratch register in the caller is set before used, there is no need to save it in the callee.

If a function save a nonscratch register that is not live after any call to this function, the save and restore of this register can be moved up to all the callers of this function. If the function is called from multiple sites, this may result in an increase of code size. However, for reasons explained shortly, this optimization turns out to be beneficial most of the time. First, if a function is called from within a loop, moving the save and restore instructions up the call graph can reduce the number of instructions executed dynamically. Second, even though the save and restore may be duplicated at the direct callers, the process does not stop there. These duplicate copies in the direct callers may again be moved to the callers of their own. In fact, the process is carried out recursively from the leaves of the call graph to the root. In some cases, all these duplicate copies eventually become one at the root, i.e., the main function.

Figure 11 shows the algorithm to move saves and restores of nonscratch registers. The algorithm does a traversal of the call graph. When a function is visited, checks are done to see if it is a library function, if its address is ever taken, or if it is already visited. If any of the above is true, the current recursion is immediately ended. Since no code for library functions is available, they are not considered in this algorithm. If the address of a function is taken, it may be called from a library function. Since in this algorithm, saves and restores of registers in a function may need to be extracted to its callers, having a library function as a potential caller prevents such an operation to be done. The last check can prevent work being done redundantly on a function called from multiple sites. More importantly, it prevents infinite recursion in the case where there are cycles in the call graph. The next step is to mark the node as visited. If the node has any children, the children are visited before any other operations are performed on the node itself. After the children are all visited, the nonscratch registers in the current node are examined in sequence. If a register is not live at any call points of the current function, the save and restore instructions of it are moved to all the callers of the current function.

As mentioned before, we mark a node upon visiting it. There is another marker used in this algorithm to handle the case of cycles in the call graph. The rest of this paragraph explains why two markers are necessary. If a function is called from two sites, then its node will have two parents in the call graph. Since after visiting a node all children nodes of it are also visited, this function will be visited by both parents. After the visit by the first parent, all registers that are not live at any call points of this function are moved up to all its callers, i.e., the two parents. By the time the second parent visits, there are no such registers left. We can either waste time examining all the nonscratch registers again, or simply end the visit by checking one marker. However, this marker does not solve all problems. Imagine the situation where the main function calls function A and function A calls function B and function B calls function A recursively. If only one marker is used, when A is visited, it is marked as such. Then B is visited, and marked as such. Then A is visited again since it is called by B. Now we see that A is already visited, therefore, go back to B and start to move registers from B to its callers, node A. Then the recursion retracts to A, the node from which B is reached. Now since all the children of A have been visited, the next step is to move some registers from A to its callers, B and main. The problem is that to move registers from A to B would be reversing the move done at B. For this reason, we use another marker to indicate whether a node has already been processed. When this marker is used for the above case and when the recursion retracts to A, we check if any of A's caller has already been performed on and find out that B has. This tells us that there is a cycle in the call graph, and we do not do anything for A. No registers will be moved from A to B or to the main function.

An improvement to the above algorithm is to rename a nonscratch register in a function when it is live at some call sites. This is possible only when another register is available for renaming. A register can be used for renaming if it satisfies two conditions:

```
PROCEDURE pushupregisters (parent)
    IF (address of parent is taken)
        RETURN
    IF (parent is marked as visited already)
        RETURN
    mark parent as visited
    FOREACH child called by parent
        pushupregisters (child)
    IF (parent is the main function)
        RETURN
    IF (a cycle is detected at this point)
        RETURN
    find all callee-saved registers in parent
    FOREACH register p saved that is live at some call points
        IF (a replacement register q not live at any call sites exists)
            replace p with the replacement q
    FOREACH register p saved that is not the pr register
        IF (p is not live at any call points)
            FOREACH caller that calls parent
                add save and restore of p to the caller
            delete save and restore of p from parent
            update callee-saved registers of parent to reflect the change
END PROCEDURE
```

Figure 11: Pseudocode for Register Optimization

1. It is not used in the function being visited.

2. It is not live at any call points of the function being visited.

The first condition is to avoid conflicts. The second condition is imposed because the purpose of renaming is to enable the save and restore of a register to be moved to its callers. The register used for renaming certainly has to satisfy the conditions for that operation.

4.2 Adding and Deleting Saves and Restores

By the convention of the *cc16k* compiler, if any register is saved in a function, the sequence of saves always starts with a decrement of the stack pointer followed by one or more saves. This initial decrement of the stack pointer takes the form of "*sp−−2*". In addition, the *pr* register, which holds the return address of a function, is usually the last one saved if it is saved at all. It is worth pointing out that when it comes to saving and restoring registers, the important thing is that they are handled in a first-in-last-out order. Which register is saved first or last is immaterial.

The fact that the DSP16k instruction set has post increment and decrement operators applied to the stack pointer makes the matter of moving saves and restores of registers a little complicated. A typical example of a save is "**sp−−2 = r4*", which saves the register *r4* on the stack and then decrements the stack pointer by two words. Because of the increment and decrement operators, in addition to adding an instruction to the code, adjacent instructions may need to be modified as well. If the added save is the only save instruction in the function, an initial decrement of the stack pointer has to be added. If there are already other registers saved in the function, then no initial decrement needs to be added since it must already be in the code. If the *pr* register is saved, the new save is added before the save of *pr*, otherwise, it is added at the end of the save sequence. This is to conform to the not-so-strict convention that the *pr* is the last register saved.

For each save instruction added to a function, one or more restores are added, since a function may have more than one return block in DSP16k assembly. Adding the restore instruction is relatively simple, since there is only one format in the restore sequence. Every restore includes an increment to the stack pointer, and there is no trailing increment like the initial decrement. For example, the instruction "*r4 = *sp++2*" restores the value of *r4*.

Figure 12 illustrates how the optimization is done with an example from test program *stats*. The part of the call graph of this application where elimination of saves and restores actually occurs is shown. None of the six functions in the graph are leaf functions, which is why they all save the *pr* register. Note that some functions called by these six functions are not shown here. In Figure 12, the registers saved and restored by each function are listed next to the node representing the function. For example, in part (1), before the optimization is applied, the registers saved and restored by the main function include *r4*, *r5*, and *pr*, and those by function *coef* include *a4*, *a5*, *a6*, *a7*, *r4*, *r5*, *r6*, and *pr*. In part (2), saves and restores of registers *a4-a7* and *r4-r6* are moved from function *coef* to the main function. We illustrate this movement with the names of these registers labeled on the arrow indicating the direction of the movement. These registers are moved because they are not live at the point in the main function where *coef* is called. After this, the only register saved by *coef* is *pr*, and the registers saved by the main function expand to *a4-a7*, *r4-r6*, and *pr*, among which, *a4-a7* and *r6* have come from *coef*. In part (3) and part (4), saves and restores of some registers are moved from *stddev* and *mean* to the main function, respectively. As a result, the number of registers saved by these two functions is reduced. However, the registers saved by the main function remain the same. This is because the main function is already saving all the nonscratch registers of interest at the end of stage (b). The saves and restores of *r4* and *r5* remain in *stddev* because these registers are live at the point where *stddev* is called from the main function and no registers are available to rename them. Next, when the algorithm visits function *rand* at

stage (e), it notices that register *r4* is live at the point where it is called from *init*. Therefore, the algorithm attempts to rename *r4* to a register that is not used in *rand* and is not live at the point where *rand* is called. Indeed, it finds register *r5* and renames *r4* to *r5* and moves the save and restore of *r5* to function *init*. These operations are illustrated in part (5), where the label "*r4→r5*" on an arrow indicates that *r4* is first renamed to *r5* and then the save and restore of *r5* are moved to the function pointed to by the arrow. Similarly, in part (6), register *r4* is renamed to *r6* and then the save and restore of *r6* are moved from *init* to the main function. A comparison of the registers saved in part (1) to those in part (6) shows that the optimization eliminated the saves and restores of 13 registers, which correspond to 26 DSP16k assembly instructions.

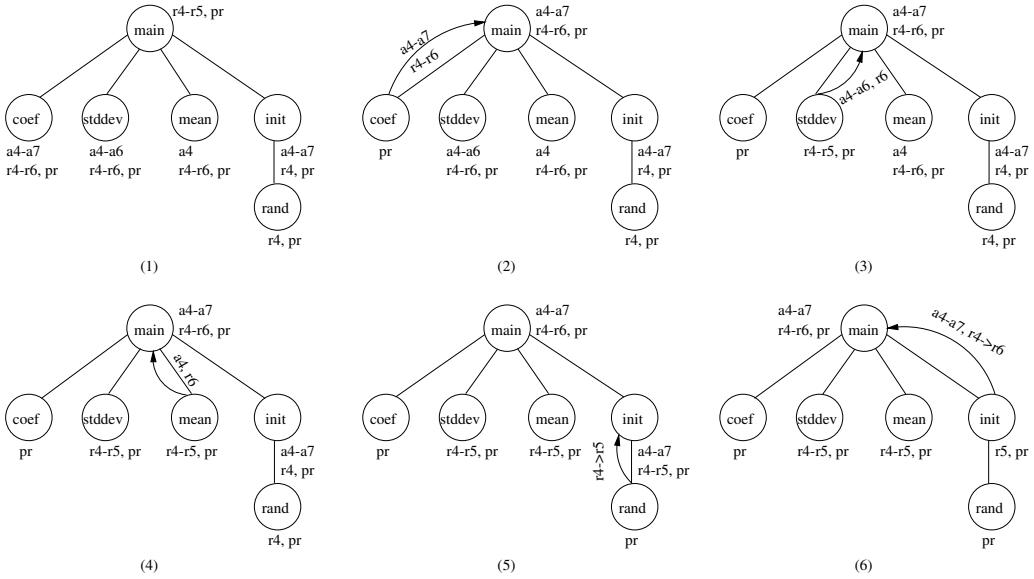

Figure 12: Example of Elimination of Saves and Restores

5 Optimization on Transfers of Control

A transfer of control (TOC) is either a branch, a jump, a call, or a return instruction. When the target of a TOC is encoded as a constant in the instruction, a larger offset of the target may require more bits. This is the case for the DSP16k instruction set, where a far TOC instruction is twice the size of a near TOC instruction. For lack of information on the offset of the target from the source, the *cc16k* compiler always generates a far TOC when faced with a choice between a far TOC and a near TOC. The goal of the optimization discussed in this section is to convert far transfers of control to near transfers of control whenever possible.

In the DSP16k instruction set, there are three different instruction sizes: 16 bits, 32 bits, and 64 bits. TOC instructions can be either 16 bits or 32 bits. Different types of TOC instructions of the DSP16k instruction set are listed in Table 1. As we can see from this table, the target of a transfer can either be a constant or an address stored in a register. This optimization is concerned with the former, since the TOC instructions with a register target are already in the minimal 16-bit format. For constant targets, the 16-bit format is used when the target offset is small enough to fit in 12 bits. The 32-bit format is used otherwise. It is therefore not surprising that the 16-bit format is also referred to as *near* transfers of control and the 32-bit format as *far* transfers of control. Another

thing that can be observed from Table 1 is that there is no 16-bit encoding for branch instructions or conditional calls with a constant target. Such instructions can only be encoded in 32 bits. As it turns out, only the instructions that begin with "*if true*" or "*far*" can be potentially reduced to 16 bits, other instructions are either already in a 16-bit format or do not have a 16-bit encoding.

Branch Instructions	**Size**	**Action**
if *condition* goto *constant*	32	none
Jump Instructions	**Size**	**Action**
goto *constant*	16	none
near goto *constant*	16	none
if true goto (pt0|pt1|pt)	16	none
if true goto *constant*	32	32→16 if offset fits in 12 bits
far goto *constant*	32	32→16 if offset fits in 12 bits
Call Instructions	**Size**	**Action**
call *constant*	16	none
near call *constant*	16	none
(if *condition*)? call (pt0|pt1|pr)	16	none
icall *constant*	16	none
tcall	16	none
if *condition* call *constant*	32	none
if true call *constant*	32	32 → 16 if offset fits in 12 bits
far call *constant*	32	32 → 16 if offset fits in 12 bits
Return Instructions	**Size**	**Action**
return	16	none
if *condition* return	16	none
ireturn	16	none
treturn	16	none

Table 1: DSP16k Transfer of Control Instructions

5.1 Transfer of Control Instructions in the DSP16k

The *cc16k* compiler uses an intermediate representation that does not always have a one-to-one mapping with assembly instructions. At the very end of the compilation, this intermediate representation is translated to assembly language. As a result, the *cc16k* compiler always uses the 32-bit format for TOC instructions with a constant target. In contrast, the optimizer reads in all the assembly files of an application and processes them together. It is possible for the optimizer to determine the distance between any two instructions since it has access to all the assembly instructions.

The algorithm to compute the distance between two instructions is expected to be straightforward if the size of each instruction in between is known. One complication comes from the fact that at the assembly code level, an instruction that requires 64 bits is actually two lines of code. It turns out that determining whether two lines of code belongs to one instruction or not is not a simple matter. The remedy is to assume that all instructions are 64 bits long. Fortunately, we are still able to convert most of the candidate 32-bit TOC instructions to the 16-bit format.

5.2 Forward and Backward Transfers of Control

A forward TOC precedes the target in the assembly file, whereas a backward TOC appears after the target in the assembly file. When calculating the distance between the source and the target of a TOC, forward transfers of control and backward transfers of control are treated differently. To see why it is done this way, lets take a look at how a constant target in a TOC instruction is encoded. As it turns out, it is encoded as the offset of the address of the target instruction from the address of the instruction following the TOC. The reason why the instruction after the TOC instead of the TOC itself is used in calculating the offset is that the program counter is incremented right after the TOC is fetched. The offset encoded in a TOC instruction represents the number of 16-bit words between the TOC and the target block. For a forward transfer of control, the offset is a positive value and it is the sum of the sizes of all instructions in between but not including the source or the target block. For a backward TOC, the offset is a negative value and the absolute value of this offset is the sum of the sizes of all instructions between and including the source and the target block. Figure 13 shows an outline of how the offsets are calculated.

```
PROCEDURE fartonear
    LET B be the list of basic blocks in the application
    FOREACH basic block cblk in B
        LET e be the last instruction in cblk
        IF ((e is a jump OR e is a call) AND
            the target is a constant AND
            (e starts with ``if true'' OR e starts with ``far''))
            LET dblk be the target basic block of e
            LET o denote the offset between e and its target in dblk
            LET r be the range of blocks that contribute to o
            IF (e is a forward transfer of control)
                r = (cblk, dblk), r is an open range
                o = sum (instruction sizes in r)
            ELSE
                r = [dblk, cblk], r is a closed range
                o = - sum (instruction sizes in r)
            IF (o can fit in 12 bits)
                replace ``if true'' OR ``far'' in e with ``near''
END PROCEDURE
```

Figure 13: Pseudocode for Far-to-Near Transformation

Figure 14 gives an example for each of the two types of transfers of control discussed in this section. The first example is a forward jump taken from the benchmark program *xc40*. The offset is given in number of 16-bit words, and each instruction is conservatively assume to be 64 bits, i.e., 4 words. An offset of 16 can easily fit in 12 bits. As a result, the far encoding is transformed into the near encoding. The second example is a backward TOC that happens to be a call instruction. The example is taken from the benchmark program *matmult*. The offset here is a negative value: -64, which can easily fit in 12 bits as well. Hence, the call instruction is also transformed into the near version.

Type	Program	Offset	Before	After
Forward	xc40	16	_.L24: a0 - 1 if ne goto _.L6 a3 = a3 if ne goto _.L30 r0 = _out_word a0 = 0x0000000a **if true goto _.L33** _.L30: a3 - 1 if ne goto _.L6 r0 = _out_word a0 = 0x0000003a **_.L33**: *r0 = a0	_.L24: a0 - 1 if ne goto _.L6 a3 = a3 if ne goto _.L30 r0 = _out_word a0 = 0x0000000a **near goto _.L33** _.L30: a3 - 1 if ne goto _.L6 r0 = _out_word a0 = 0x0000003a **_.L33:** *r0 = a0
Backward	matmult	-64	**_InitSeed:** sp−−2 *sp = r6 r6 = sp+0 r0 = _Seed a0 = 0 *r0 = a0 _.L2: sp = r6 nop r6 = *sp++2 nop if true return _main: sp−−2 *sp−−2 = r6 *sp = pr r6 = sp+0 **far call _InitSeed**	**_InitSeed:** sp−−2 *sp = r6 r6 = sp+0 r0 = _Seed a0 = 0 *r0 = a0 _.L2: sp = r6 nop r6 = *sp++2 nop if true return _main: sp−−2 *sp−−2 = r6 *sp = pr r6 = sp+0 **near call _InitSeed**

Figure 14: Examples of Far-to-Near Transformation

6 Results

Five test programs are used to measure the effectiveness of the four optimizations discussed in this report. Table 2 gives a brief description of these test programs. The results of code size are listed in Table 3, and those of execution cycles are listed in table 4. The first column of each table gives the name of the test programs. The second column gives the results after dead code removal is applied. The third column lists the results after inlining is applied in addition to dead code removal. In the fourth column, one more optimization, namely, eliminating unnecessary saves and restores of registers, is added. In the last column, all four optimizations are performed. The percentage is measured against the effectiveness of the optimizer without using any of these four optimizations.

Program	Description
edn	encoding and decoding
fft	128 point complex fft
stats	statistics program
matmult	matrix multiplication
total8	image processing

Table 2: Description of Test Programs

As can be seen from the numbers, the optimizations are very effective in reducing the code size, particularly due to dead code removal which deletes complete functions that are kept by the *cc16k* compiler after inlining but are not called from anywhere else. Inlining functions called from a single site also has a positive yet small effect on reducing code size. Removing unnecessary saves and restores can have a significant effect in some cases. For example, it resulted in an additional reduction of 5.26% in code size when applied to *stats*. The effect of far-to-near transformation in reducing the code size is small yet it is performed very often due to the commonness of transfer of control instructions.

Program	dead code removal	+ inlining	+ remove save/restore	+ far-to-near
edn	-42.00 %	-42.16 %	-42.93 %	-42.93 %
fft	-6.21 %	-7.45 %	-8.94 %	-9.07 %
stats	-2.92 %	-5.46 %	-10.72 %	-11.89 %
matmult	-44.62 %	-46.22 %	-46.02 %	-46.02 %
total8	-59.32 %	-59.32 %	-59.32 %	-59.32 %

Table 3: Results of Code Size

The effect of these optimizations on reducing the execution cycles is not as noticeable. One of the reasons is that even though dead code takes up space, it does not slow down execution because it is not executed anyway. A near transfer of control takes less space than a far transfer of control, but it takes just as much time to execute. Therefore, no reduction in execution cycles is expected from these two optimizations. Inlining functions can make it possible to place a loop in the ZOLB, in which case the execution time can be reduced significantly. However, caching loops is not part of this project, therefore, its benefit is not discussed in this report. Interested readers can obtain relevant papers from Dr. Whalley. Both function inlining and elimination of saves and restores result in deletion of instructions, therefore, some degree of reduction in execution cycles is expected from them. The effectiveness of these two optimizations on reducing execution cycles depends very much on the program to which they are applied. To see any effect at all, the application has to

provide opportunities for functions to be inlined and for saves and restores to be eliminated. For this reason test programs consisting of only the main function are not used. When elimination of instructions occur in an inner loop of a large number of iterations, the effect is expected to be more obvious. However, no detailed analysis in this respect is done due to two reasons. First, the only way to get measurements of execution cycles off-site is through the use of a simulator, which runs many orders of magnitude slower than the real DSP16k chip. A loop of a large number of iterations can take hours to execute. A more important reason is that the measurements of execution cycles are not always accurate in the simulator.

Program	dead code removal	+ inlining	+ remove save/restore	+ far-to-near
edn	-0.80 %	-0.01 %	-0.01 %	-0.10 %
fft	-0.00 %	-0.00 %	-1.96 %	-1.96 %
stats	-2.37 %	-0.03 %	-2.40 %	-2.40 %
matmult	0.00 %	-7.62 %	-9.26 %	-9.26 %
total8	-0.26 %	-0.26 %	-0.26 %	-0.26 %

Table 4: Results of Execution Cycles

7 Conclusions

The goal of this project is to use information that is not available to the *cc16k* compiler to perform interprocedural optimizations. Four such optimizations are: removing dead code, inlining functions, eliminating unnecessary saves and restores of registers, and converting far transfers of control to near transfers of control.

Dead code removal and far-to-near transformation are intended to reduce code size. Dead code removal proves to be very effective when the *cc16k* compiler inlines functions and keeps the functions inlined even when they are not called elsewhere. Opportunities for far-to-near transformation exist in most applications due to the commonness of transfer of control instructions.

Inlining functions to make loops cacheable provides opportunities for caching loops which eventually reduces execution cycles. When this optimization is applied, code size may be increased if a function inlined is called from more than one site.

Inlining functions called from a single site and elimination of unnecessary saves and restores of registers are expected to reduce code size directly and to decrease execution cycles as the result of fewer instructions executed. The degree to which these two optimizations reduce code size is usually small. The effectiveness of these two optimizations in reducing execution cycles is dependent on the dynamic execution of the application. If instructions from a loop of many iterations are removed as the result of these two optimizations, the effectiveness is expected to be obvious.