FUTURE COMPILATION REQUIREMENTS FOR EMERGING DRIVING GENERAL PURPOSE APPLICATIONS

BY

IAN STEINER

B.S., University of Illinois at Urbana-Champaign, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

As industry moves from single processor systems to chip multiprocessors in the general purpose community, it is becoming increasingly important for research to help enable this transition by developing tools that assist programmers in developing applications for these systems. Compilers will play an important role in this transition. There has been a wealth of past research developing compilation tools to enable high-performance computing, which has utilized multiprocessor systems for years. However, there is a gap between these applications and the general purpose "driving" applications of the future. This thesis will provide an evaluation of four representative benchmarks and provide insights into what new research needs to be completed in order to extend past work to these future applications.

Much of the research in parallelizing compilers has focused on scientific applications that are rich in scalable inner loop parallelism. Many of these applications are written using Fortran and perform repetitive calculations on large arrays of data. While C implementations exist, these applications do not take advantage of many of the features available in C. They are similar in structure to their Fortran counterparts and therefore do not suffer the same complications as general purpose applications.

Future general purpose applications exhibit some of the characteristics of scientific applications. This thesis contends that these applications will generally model the physical world, which naturally contains large amounts of inherent parallelism. However, unlike past scientific applications, these applications commonly take advantage of the additional features provided by C, including complicated memory usage patterns and large code bases that perform a bulk of similar tasks. By evaluating the transformation and analysis requirements for attaining parallel implementations of four representative benchmarks, this thesis motivates important research problems for the IMPACT compiler for enabling this important transition in general purpose computing.

ACKNOWLEDGMENTS

I would first like to thank Wen-mei Hwu for guiding me through my final three years at the University of Illinois. He provided me with excellent insights into both research and the world in general.

I would like to thank all of the past members of IMPACT for their tireless work building the compiler infrastructure that I used on a daily basis. I would especially like to thank John Sias for mentoring me in both research and life through the years; Hillery Hunter and Erik Nystrom for their additional guidance and support; Sain-Zee Ueng and Jame Player for their friendship and collaboration; and Shane Ryoo, Bob Kidd, and Chris Rodrigues for their collaboration. This thesis incorporates the hard work of many people. I would in particular like to thank Chris Kung for his help on *jpegdec*, Sara Sadeghi for her help on *mpg123*, and John Stratton for his help on *LAME*. Steve Lumetta and Matt Frank (UIUC) supported my research by providing me with guidance and insight into my work. I also learned much from Yefim Shuf (IBM) during our collaborations on performance analysis.

Thanks to the Gigascale Systems Research Center and the Department of Electrical and Computer Engineering for their support, both monetary and otherwise, over the years.

Finally, thanks to my parents, my sister, and my fiance for their support and love. Without them I never would have made it this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Since its creation in 1988, the IMPACT compiler has focused primarily on Explicitly Parallel Intstruction Computing (EPIC) compilation through the extraction of Instruction Level Parallelism (ILP). Many of the compilation and architectural techniques developed with this infrastructure have had a significant impact on industry architecture, recently culminating in the creation of the Itanium 2 processor. In [1], Sias et al. evaluated many of the techniques developed over the years, including superblocks, hyperblocks, and other predication techniques, and summarized much of the IMPACT EPIC ILP research.

This past work focused on improving sequential application performance on single processor systems. For years, much of the improvement in processor performance has been driven by transistor sizing, which has allowed for faster clock rates and more complicated designs. While scaling down the technology size has always been a challenging task, the International Technology Roadmap for Semiconductors [2] predicts that it will not be possible to continue to follow Moore's Law.

As transistor sizes have shrunk in recent years, architects have had a difficult time making use of the additional resources because of the physical limitations of silicon. Power density, heat, and leakage have become major roadblocks preventing more complicated monolithic core developments. As such, industry has started to return to simpler designs; Intel for example is basing their latest chips off of the Pentium 3 architecture and not their more recent Pentium 4.

So where should industry go from here? When designing processors, and products in general, it is important to understand what needs they are intended to fulfill. With processors, one must understand the characteristics of the software that will drive innovation in the future. This thesis contends that software that models the physical world will be the "driving applications." The physical world, and the applications that model it, is rich with inherent parallelism. This is convenient, as industry is in the process of developing chip multiprocessor (CMP) solutions. It was difficult, if not impossible, to take advantage of multiprocessor systems with the sequential applications that have driven innovation over the past decade. However, as these new driving applications begin to step into the spotlight, there will be a wealth of opportunities for CMP systems. There are many difficult research problems involved in developing both the hardware

and software infrastructure that will be used as industry moves away from single processor systems.

This thesis investigates the characteristics of four applications that reflect future driving applications. It will provide overviews of the applications at an algorithmic level in addition to their sequential implementations. It will investigate possible parallel implementations, while evaluating the necessary transformations and analysis for extracting parallelism from the original sequential versions. Finally, it will provide a general road map for future developments of the IMPACT compiler. Chapters 2 and 3 provide background information and an overview of the IMPACT compiler. Chapter 4 evaluates four benchmarks: *jpegdec*, *LAME*, *mpg123*, and *179.art*. The thesis concludes with a summary of the findings and a roadmap for future development in Chapter 5.

CHAPTER 2
BACKGROUND

Before investigating some future driving applications, this chapter provides some necessary background and terminology that will be used throughout the remainder of this work. One of the major goals of this thesis is to provide new students with the necessary background to make contributions as quickly as possible; as such, this section is targeted at those individuals who are new to the compiler and parallelization communities. Chapter 3 will provide an overview of the IMPACT compiler as it pertains to parallel compilation.

2.1 Preliminary Terminology

Two instructions that write to the same memory location or register are said to have an *output dependence* between them. Additional common dependence terminology is defined in [3]. In addition to these common dependences, compilers contain dependences that do not actually exist in the application. Compilers must guarantee correctness, and this results in conservative analysis frameworks that are unable to generate perfect dependence information – especially in complicated applications. This thesis will use *spurious dependence* to refer to those dependences that the compiler identifies that do not actually exist in the application. Parrallelizing compilers must reduce the set of spurious dependences in order to be successful, and this will be the target of much future research.

Applications tend to have multiple possible parallel implementations. This thesis will refer to parallelism that encompasses relatively small amounts of computation as "fine-grain," and that which encompasses large amounts as "coarse-grain." Course-grain parallelism is more desirable for extracting thread-level parallelism because such implementations better amortize the costs of extracting threads. In the applications studied, it is generally not possible to extract course-grain parallelism from the inner loops of applications.

2.2 Figure Conventions

Figure 2.1 is an example of the figures that will be used in Chapter 4 to illustrate the important data and control flow in sections of code. It also presents five conventions that will be followed in these figures. These conventions are not restated in the text. In this example,

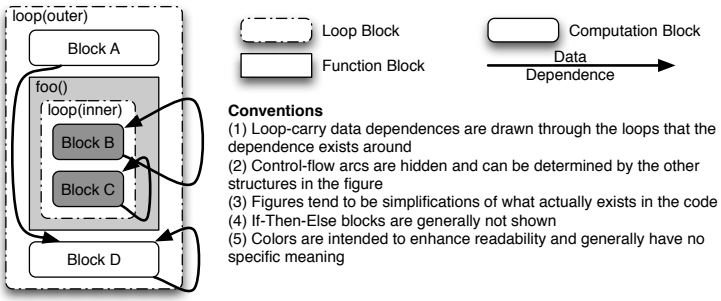


Figure 2.1 Application Parallelism Figure Conventions

Block A is inside of the outer loop. The function foo() is called from the outer loop and this call follows the execution of A and precedes the execution of Block D. Block A has no loop-carried dependences or input dependences in this scope. Block D contains an input dependence from Block A along with a loop-carried dependence from itself around the outer loop. Inside of the function call is an inner loop that contains two blocks of computation, B and C. Block B has a loop-carried dependence around the outer loop, while C has one around the inner loop. Some figures will not exactly follow this convention, and will be labeled accordingly.

2.3 Parallel Compilation

Scientific computing has been making use of multiprocessor systems for many years. There has been extensive research into automatic parallelization techniques in the High Performance Computing (HPC) community that have targeted these scientific applications. Computationally intensive inner loops that process large arrays of data with a substantial inherent parallelism are common in these applications. Scientific applications are commonly written in High Performance Fortran (HPF) [4], which does not have the notion of pointers like C and C++. Many powerful transformation and analysis techniques were developed for this paradigm [5], and provide the foundation for the ongoing work in the IMPACT compiler. It is imperative for developing parallel compilation infrastructures to understand, incorporate, and build upon this extensive past work. In Chapter 4, *179.art*, a C benchmark from SPECfp2000 [6] that falls into this category of applications, will be evaluated.

2.3.1 Loop transformations

In order to extract high-performance parallel code from HPC applications, much work was done on loop transformations that reordered code execution. Banerjee's book [7] provides a

reference for many of these transformations as well as the mathematics behind them.[1] This section provides an overview of three important types of loop transformations that will be encountered in Chapter 4. While this thesis will focus on extracting thread-level parallelism, these transformations are useful for instruction and vector level parallelism as well.

Figure 2.2 illustrates three important loop transformations: (a) *loop fusion*, (b) *loop fission* (or *loop distribution*), and (c) *loop interchange*. The first takes two (or more) loops and "fuses" them into one, the second takes a single loop and reconstructs it into two (or more) separate loops, and the third "interchanges" different loops in a loop nest.

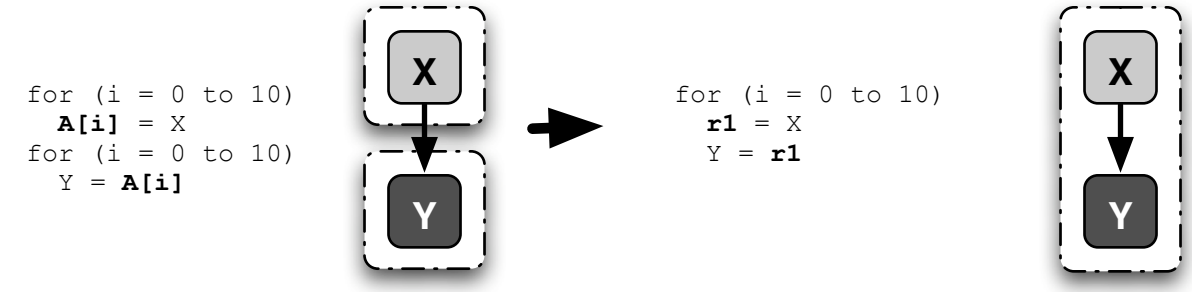
Loop fusion, which combines two or more loops into a single larger loop, is used to increase the size of parallel blocks of code. Figure 2.2(a) illustrates this transformation. In any parallel paradigm, there is generally overhead to performing parallel execution of loops which can be better amortized by increasing the granularity. The figure also shows that loop fusion can be used in conjunction with register promotion to reduce memory traffic. If A[] is not live out the bottom of the second loop, and only is used to communicate between the two loops, then both a load and a store can be removed. If A[] is potentially live out, then the store must remain but the load can still be removed.

Loop fission, also commonly called loop distribution, takes a single loop and splits it into two (or more) loops over the same iteration space. This is the opposite of loop fusion. In the left diagram of (b), there is a loop-carried dependence that serializes the computation. By splitting the loop into two separate loops (as shown on the right), the A block can be executed in parallel by performing *scalar expansion* and creating an array for each iteration's value of r1. Increased memory traffic is traded for increased parallelism. Loop fission is useful for removing *recurrences* or dataflow strongly connected components (SCC) from loops, particularly in cases where the majority of the computation is parallel and outside the SCCs, allowing for additional parallelism to be extracted. In the example, accesses to B[] in each iteration depend on the previous iteration because of the i-1 index. The multiplication operation is independent from iteration to iteration. By separating the loop on the left into two separate loops, the multiplication iterations can be performed in parallel.

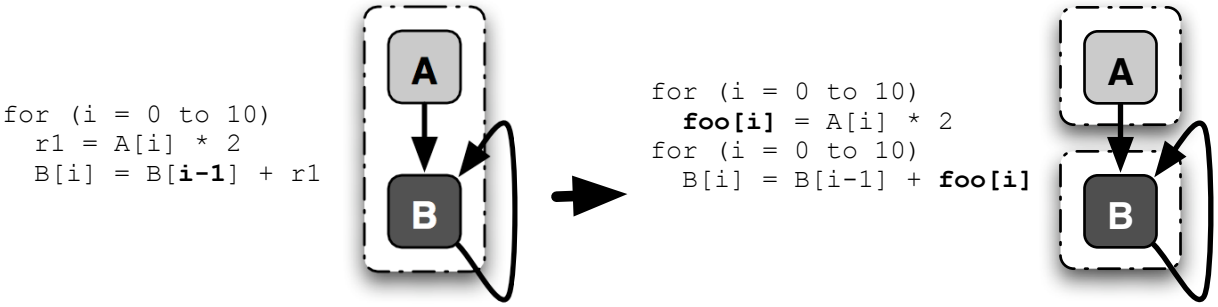
Finally, loop interchange exchanges the order of nested loops. This can be used to move the location of loop-carried dependences.[2] In the left diagram in (c), there exists a loop-carried dependence in the inner loop (because of the j-1) and no dependence around the outer loop.

[1] A large portion of this work was led by Prof. David Kuck at the University of Illinois as well as Prof. Ken Kennedy at Rice University.

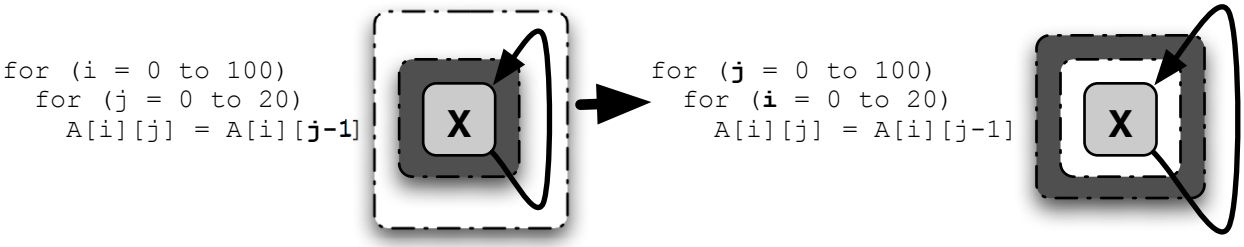
[2] It can be used for other optimizations as well, including memory-locality optimizations.



(a) Loop Fusion converts two separate loops into a single loop. This can be used to:
(1) improve the memory characteristics (`A[]` can be removed if it is not live out of **Y**)
(2) increase parallelism granularity



(b) Loop Fission (or Loop Distribution) converts a single loop into two (or more) separate loops with the same iteration count. This generally trades off increasing the memory usage (note that the foo array has been added) for increased parallelism. The **A** block of code has no loop carry dependences, and therefore can be executed with a high degree of parallelism.



(c) Loop Interchange changes the order of loop nests in an attempt to move the loop-carry dependence. In the example, there exists a loop-carry dependence through the inner loop that prevents extraction of inner loop parallelism. By interchanging the loops, the inner no longer contains a loop-carry dependence. One can exchange loops in either direction depending on the type of parallelism that is desired.

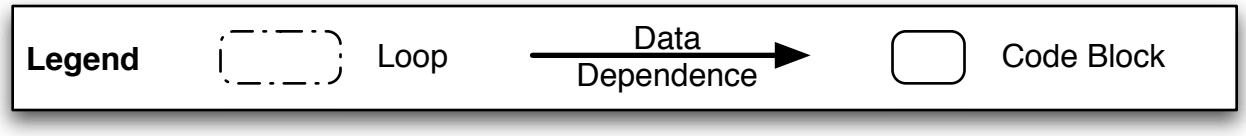


Figure 2.2 Loop Transformations

This is illustrated in the figure with the dependence being drawn through only the first loop block. By interchanging the two loops, the loop-carried dependence is moved to the outer loop, resulting in inner loop iterations that can be executed in parallel. If outer-loop parallelism is desired, then opposite interchange could be performed on loop nests with parallel inner loops.

2.3.2 Privatization

In order for threads to be extracted from sequential code, certain variables must be *privatized* in order for the computation to work. For example, if variables are declared at the top of a function, and are used in loop nests inside that function that are transformed into threads, those variables must be replicated for each of the threads for proper execution. This situation extends beyond scalars allocated on the stack to all types of data.

Privatization will be an essential component of the future IMPACT work. It has been studied extensively in the past [8], and many of these techniques will have to be utilized in order complete parallel versions of these applications.

2.3.3 Analysis

There are two significant types of analysis that currently exist in the IMPACT framework. Pointer analysis, which is used to disambiguate different memory objects, will be covered in detail in Section 3.2.2. After pointer analysis is performed, additional *array disambiguation* analysis can be performed to improve the dependence resolution on array objects. These analyses are commonly built using *Single Static Assignment* (SSA) representations (Section 2.3.3.1). One example of such an analyses is the Omega Test [9] (Section 2.3.3.2). While these two types of analysis generally focus on reducing the set of spurious dependences, this thesis will demonstrate the need for other analysis techniques that are not directly related to dependencies.

2.3.3.1 Single Static Assignment

In superscalar processors, register renaming is used to dynamically rename different versions of registers in the dynamic instruction stream. It is useful for compilers to perform a similar renaming process when performing analysis. SSA [10] provides a representation of an application's dataflow that renames different versions of variables (or registers in the case of Lcode or assembly). In addition to creating different versions for variable from code like `a = a + 1`, SSA also inserts ϕ-nodes at control flow points where different versions of variables join. For example, if there is an `if` block in code that contains an assignment, a ϕ-node would be

inserted below the block. This would signify that the version of the variable below that point could either come from the definition within if block, or the previous definition. Similar cases exist at the top of loops and other control-flow join points.

There exist other more detailed extensions to SSA. SSA itself is not an "executable" representation as it does not maintain all of the control flow information that is necessary for the proper execution of an application. *Program Dependence Graphs* (PDG) [11] and *Value Dependence Graphs* (VDG) [12] both are extensions of SSA that incorporate the remaining control flow information and result in an executable form. These extensions are useful because transforming them handles both the necessary data and control considerations. IMPACT currently contains SSA implementations in both Pcode and Lcode (Chapter 3) that, when combined with the control flow graphs (CFG), provide all the information contained in a PDG or VDG, but not in a single executable form. *Gated Static Assignment* (GSA) [13] is another extension of SSA that contains different types of nodes for different types of control-flow join points. These nodes also contain additional information about their control flow. GSA currently is not implemented in IMPACT, but extending the existing infrastructure would not be difficult if it was determined to be necessary.

2.3.3.2 Array disambiguation - Omega Test

While pointer analysis is able to distinguish different memory operations that refer to different memory objects (Section 3.2.2), dependencies between accesses to arrays can be broken if it can be proved that they are to distinct indices. For example, in the loop for (i = 1 to 10) {a[i] = a[i] * 2;}, although each iteration accesses the same array, each iteration is independent because the indices *i* are always distinct. Omega Test [9] is a tool that is capable of distinguishing between different array memory accesses in loops. Loops contain *induction variables* that change in a predictable manner for each loop iteration. For example, in the above loop, i is an induction variable. The *fundamental induction variable* in a loop refers to a conceptual induction variable that starts at 0 and increments by 1 for each loop iteration. The other induction variables in loops can be expressed as a function of the fundamental induction variable. Equations that express array reads and writes in loops as functions of the fundamental induction variable (or variables in the case of multidimensional loops) are called *induction expressions*.

After calculating induction expressions for the different array accesses in a loop, a set of linear equations can be generated based on these induction expressions and the loop bounds. Solving these linear equations not only removes some dependences, but can also provide more detailed information about dependences that may exist. For example, it may be possible to

determine the *dependence distance*, or number of iterations between which loop-back dependences do not manifest themselves. For example, in for (i = 0 to 10) {a[i] = a[i - 2] * 2;}, there is a dependence distance of 2 because loop iteration n depends on the result of iteration $n - 2$.

Additional work has been done to extend the concept of Omega Test to handle certain special cases. Omega Test did not originally support affine expressions that incorporated undefined invariants. Many of the induction expressions contain constants (integers) multiplied by an induction variable. I-Test [14] extends Omega Test by allowing for loop-invariant variables to be used in addition to integer constants.

Shape Analysis [15, 16] is a technique that attempts to identify other types of data structures in applications, and then applies the semantics of those structures to reduce the spurious dependencies. This type of analysis will be necessary when working with applications that are not limited to array-based structures.

CHAPTER 3
IMPACT COMPILER OVERVIEW

While Chapter 2 provided an overview of parallel applications and compilation, this chapter will provide details about the existing IMPACT features. It will not present the ILP compilation optimizations that have been implemented in past work [17], but focus on the components that are relevant to thread-level parallel compilation. Basic compiler knowledge, as presented in [8, 18], is assumed.

3.1 High-Level IMPACT Overview

The IMPACT compiler has two internal representations, *Pcode* and *Lcode*. EDG, a commercial parser and lexer, processes the input C, C++, or Fortran code, and this is then translated into Pcode, an Abstract Syntax Tree (AST) representation. After several analysis and transformation steps, Pcode is translated into Lcode, a register transfer level representation that is similar to assembly code. The majority of past analysis and optimization research was performed here. IA-64 assembly code can be generated from Lcode, or it can be converted back into C for compilation to different architectures.[1]

3.2 Pcode

Pcode performs three important tasks: profiling, inlining, and pointer analysis. Pinline uses the profile information and heuristics based on profile weights, function sizes, and code growth, to decide which function calls to inline, and is capable of inlining direct, indirect, and recursive function calls. Pcode also incorporates a scalable context sensitive, partial flow sensitive, field sensitive pointer analysis based on Erik Nystrom's [19] and Jame Player's [20] theses.

3.2.1 Pinline

Extending past HPC analysis and transformation techniques to loop nests that contain procedure calls is nontrivial. Because IMPACT does not currently have an interprocedural analysis

[1] Other code generators existed in the past, but were retired from lack of use. There also exists a code generator for the IMPACT ISA that is used in the internal Linterpret simulator.

and transformation framework, it is necessary to use Pinline to selectively inline the necessary functions. IMPACT contains a runtime compilation parameter (PARM) that inlines all function calls (with the exception of recursive calls[2]). Pinline also obeys the inline function directive. Functions that are declared with the inline directive, including, to a limited extent, those that are called indirectly, will be inlined at all call sites. In gcc, the inline directive traditionally was only obeyed for functions declared statically in the same file. Pinline can and will inline functions across files. Pinline handles indirect function calls by inserting control flow to check for a specific case and then inlining that case within the control flow. Indirect calls are never removed, but one or more special cases can be executed instead.

Inlining is performed at the beginning of the compilation process. As a result, any analysis that is developed can take advantage of the inlined code. However, Pinline is also forced to base its decisions on profile data and not analysis results. For example, pointer analysis will determine the set of potential callees for indirect function calls. Because Pinline is run first, it is not possible to use this information, and only callees that are exposed during profiling can be inlined.

3.2.2 Pointer analysis

One of the major differences between HPC and general purpose applications is the usage of heap-allocated objects. Pointer analysis is the first step in disambiguating memory accesses to these objects. Traditionally, pointer analysis in IMPACT has been used to disambiguate stores and loads and to perform optimizations inside small regions of code. Successfully removing as many spurious dependences as possible to reduce the problem set is important and necessary in bridging the gap between scientific and general purpose applications. Inaccurate pointer analysis results may either result in suboptimal parallel compilations or, potentially, prevent parallel extraction altogether.

IMPACT's FULCRA framework for pointer analysis uses the notion of *points-to sets* to present its results. Different pointer analysis *objects* are created, and each memory operation is tagged with the objects that it *may* access using *access specifiers* or *accspecs*. Two different memory operations are said to *alias* if there exists an intersection in their points-to sets. FULCRA incorporates scalable context and field sensitive pointer analysis with heap cloning [19]. Support for partial flow sensitivity using SSA [20] can be enabled for the input to FULCRA,

[2]Recursive calls can and will, to a limited extent, be inlined as well. For example, if it can be determined that a certain recursive call is almost always performed *n* times, then it may be inlined up to that point.

but the results are not currently annotated onto the resultant Pcode. High-level descriptions of these features follow.

3.2.2.1 Context sensitivity

Context sensitive pointer analysis distinguishes between call paths when determining how callees affect callers. For example, if functions A and B both call X, a context insensitive analysis may show data flow from A to B through X. This is because the summary of X that is used will merge information from each call site. Context sensitivity avoids merging by separating different call sites. While it is possible to achieve context sensitivity by simply inlining everything, that is not a scalable solution. FULCRA provides a more scalable context sensitive framework [19].

3.2.2.2 Heap cloning

Heap cloning is a feature that allows the compiler to *version* heap objects based on the call stack that they are allocated from. For example, it is common to create allocation routines for different objects in C. In basic pointer analysis, pointers that are allocated from the same malloc call are assigned to the same initial pointer-analysis object; consequently any objects allocated by the same allocation routine will alias. By considering the call stack, heap cloning emulates the inlining of the different allocation routines to disambiguate the different objects.

Version information is stored along with each object ID in the accspecs of each memory operation. Accesses where heap cloning was not used will be marked as such, and cannot be disambiguated using version information. Accesses to different versions of the same object do not alias. Currently, the versioning system is exclusively used for heap cloning, but it is possible to use it for other future analysis results.

3.2.2.3 Field sensitivity

In C applications, it is not uncommon to malloc structures that contain pointers to other objects. In these situations, the pointer to the outer structure contains at least two levels of indirection to the actual data. Field sensitive analysis distinguishes between the objects pointed to from different fields inside structures, while basic pointer analysis will merge these accesses. It is important to note that field sensitivity does not refer to the case of distinguishing between accesses to different fields of a structure which are always independent.

```
 1 typedef struct
 2 {
 3   int *ptr1;
 4   int *ptr2;
 5 } structure;
 6
 7 void main()
 8 {
 9   structure *A;
10   int *p1, *p2;
11
12   /* allocate data on heap */
13   A = (*structure) malloc (sizeof(structure));
14   A->ptr1 = (int*) malloc (10 * sizeof(int));
15   A->ptr2 = (int*) malloc (10 * sizeof(int));
16
17   /* retrieve pointers from structure fields */
18   p1 = A->ptr1;  p2 = A->ptr2;
19
20   /* dereference pointers from structure fields */
21   *(a->ptr1) = 1; *(a->ptr2) = 2;
22 }
```

Figure 3.1 Field Sensitivity Example Code

Figure 3.1 presents example code for where field sensitivity is useful. In this example, A is a pointer to a structure in memory that contains two integer pointers. The two integer pointers are allocated at separate malloc sites, and do not alias. At line 18, the two loads can never alias because they are to different offsets in structure,[3] and field sensitivity is not necessary to handle this case. It does handle the stores at line 21. A field sensitive pointer analysis will keep the objects pointed to by ptr1 and ptr2 distinct, and will determine that these stores will never be to the same memory location. Field sensitive analysis removes the assumption that pointers contained in different fields of a struct may alias.

3.2.2.4 Partial flow sensitivity

Basic pointer analysis does not take into account the execution order of an application. All instructions are thought to execute simultaneously. In some situations, pointer variables have disjoint lifetimes that exhibit different points-to sets, and other situations where the flow through an application may prevent different pointers from aliasing. Flow sensitivity [21] targets these problems and attempts to incorporate execution order into pointer analysis. Traditional flow sensitivity tends to be expensive. IMPACT contains a partial flow sensitivity framework [20] that addresses the disjoint lifetime situations by using SSA (Section 2.3.3.1).

[3] Determining that loads do not alias is generally not useful. They are used in this example for simplicity.

By renaming variable lifetimes, it is possible to achieve higher resolution on variables that have disjoint lifetimes. As reported in [22], full flow sensitivity may not always be useful, as many of the important cases are handled by context sensitivity.

3.3 Lcode

Lcode has traditionally been the part of the compiler where, with the exception of the pointer analysis work, the IMPACT group's research has focused. It performs the traditional optimizations that one would read about in the Dragon Book[4] [18]. It also incorporates the many ILP transformations that were developed over the years for EPIC compilation [17].

After the Pcode to Lcode conversion, the traditional optimizations are performed in Lopti, followed by Lcode profiling. Traditionally this was followed by the EPIC optimizations (superblock (Lsuperscalar) and hyperblock (Lblock) formation) and then scheduling and code generation (in the Schedule Manager (SM) [23] and Ltahoe). C code can be generated at any time using the Lemulate module.

3.3.1 Lcode SSA

Lcode's core infrastructure contains support to build an SSA graph [10] off of the default internal representation (IR). In addition to this, it contains a module, Lssaopti, that performs optimizations and analysis based on this SSA graph. This is currently the site of many of the array disambiguation analysis techniques that are under development for the parallelization effort, including Omega Test [9] (Section 2.3.3.2) and I-Test [14]. While SSA provides a useful structure for performing analysis, transformations made on the SSA graph are not reflected in the Lcode IR. Therefore, transformations should generally be written for the standard IR based on the information provided by SSA-based analysis.

A previous version of pointer analysis used *sync arcs* instead of the points-to sets and accspecs that are used today. These arcs existed between different memory operations to mark dependences. The functionality for these arcs still exists today, and they are used to provide additional information to the accspecs. For example, Omega Test uses sync arcs to mark the dependence distance between different array accesses. This is a useful technique for the near term, but may need to be re-evaluated for future scalable solutions.

[4]There is a picture of a dragon on the front of [18], and it is commonly referred to by this name.

3.3.2 Lcode transformation

Two new Lcode modules, Lpar and Ltrans, have recently been added to support thread-level parallel transformations. This will be the site of many of the loop transformations explored in Section 2.3.1. Lpar contains support to transform loops with independent iterations into loops that spawn threads to perform the iterations in parallel. *Loop Blocking*, which transforms singly nested loops into multidimensional loops, was recently implemented in Ltrans. This transforms a single loop that iterates n-times into two nested loops that iterate m and $\frac{n}{m}$ times each. The order of execution is not modified in this transformation. Loop fission is currently under development in Ltrans.

3.3.2.1 IMPACT threads

Lpar spawns threads using the IMPACT thread library, *ithreads*. The ithreads library contains features similar to pthreads, but leaves some control over the implementation to IMPACT. The Liberty Simulator Framework (LSE) [24], developed by David August's group at Princeton, is one target for future studies. Pthreads requires the support of the operating system's scheduler. Because LSE is not a full-system simulator, the system calls that pthreads utilizes are not supported. However, it is possible to use *cloned threads*, which are based on the supported *clone* system call. Depending on the target system, ithreads allows the use of either pthreads or cloned threads without changing the compilation parameters.

3.3.3 Lemulate: Lcode to C

Lcode can be translated into C code for compilation by gcc (or other compilers) for different architectures. Each Lcode statement is translated into a corresponding C statement. Traditionally, particularly for the EPIC compilation work that depends on strong analysis and scheduling algorithms, this was not a high-performance solution. When converted to C, all memory dependence information is lost. The bulk of the benefit from thread-level compilation is derived from coarse-grained transformations. Converting such transformed code to C for compilation on different architectures provides an acceptable means for evaluation. This path can also be leveraged for collaboration efforts with other research groups that already use C.

CHAPTER 4
BENCHMARKS

This chapter will investigate the opportunities for future research in parallelizing compilers using the *jpegdec* image decoder, the *LAME* mp3 encoder, the *mpg123* mp3 decoder, and *179.art*, a tool for identifying thermal images inside of other thermal images. Each section will contain an overview of the benchmark, followed by an evaluation of some different opportunities for parallelism, and finish with a detailed study of some of the analysis and transformation techniques that will be necessary for future compilation frameworks in order to extract the different granularities of parallelism.

4.1 *jpegdec* - Image Decoder

The *jpegdec* application performs compressed image decoding on all types of JPEG images. There is no single standard for how JPEG images are compressed, and many different options must be supported by a decoder. At a high level, *jpegdec* first performs decompression, then inverse discrete cosine transform (iDCT), upsampling, and finally color conversion. Because there are a variety of techniques for performing each of these steps, *jpegdec* is implemented with a series of indirect function calls that are configured once at the beginning of the execution based on both the command-line parameters (for the output image type) and the image header (for the input image type). This application is similar to the *jpegenc* application, as it performs the inverse operations and exhibits similar characteristics.

This study will focus on a single popular runtime flow for decoding JPEG images (Table 4.1). This will motivate the need for path-specialized analysis and transformation, and demonstrate how runtime dead code makes automatic parallelization difficult for both analysis and transformation.

In general, the JPEG decompressing algorithm is highly parallel in some areas and less so in others. In this configuration, the input image has been compressed using Huffman encoding. Huffman decoding is inherently sequential – one needs to decode each key before it is possible to decode the rest of the string. The other portions of the decoding process are all parallel at different levels. For example, YCC to RGB color conversion is done independently for each pixel in the image, while the iDCT works on 8x8 blocks of samples (subcomponent of a pixel).

Table 4.1 *jpegdec* Runtime Parameters

Component	**Mode**	**Function**
Decompression	Huffman Decode	decode_mcu()
iDCT	Slow but Accurate Integer iDCT	jpeg_idct_islow()
Upsample - Y	Fullsize	fullsize_upsample()
Upsample - Cr, Cb	H2V2 Fancy	h2v2_fancy_upsample()
Color Convert	YCC to RGB	ycc_rgb_convert()

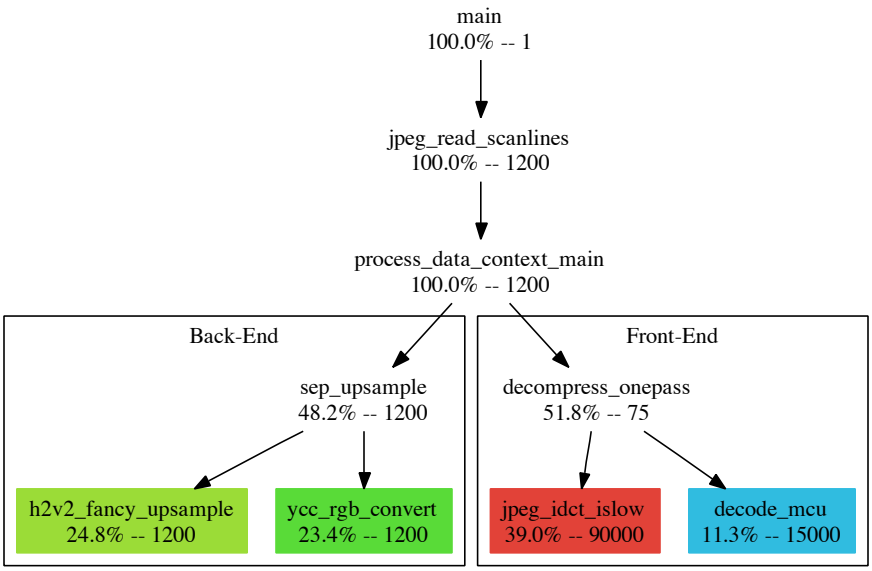


Figure 4.1 *jpegdec* Callgraph with Runtime Weights

Figure 4.1 shows the pruned callgraph of *jpegdec* with runtime weights for all functions with more than 10% of the runtime.[1] As shown, there are essentially four major components to this application (the four bottom boxes), and there is no single component that contributes to the majority of the runtime. As such, Amdahl's law prevents significant performance improvements from being gained without targeting the entire application. Each of these components demonstrates different parallelization opportunities, each with different characteristics and benefits.

[1] Profile information was collected using gprof on an AMD 2800+ x86_64 system.

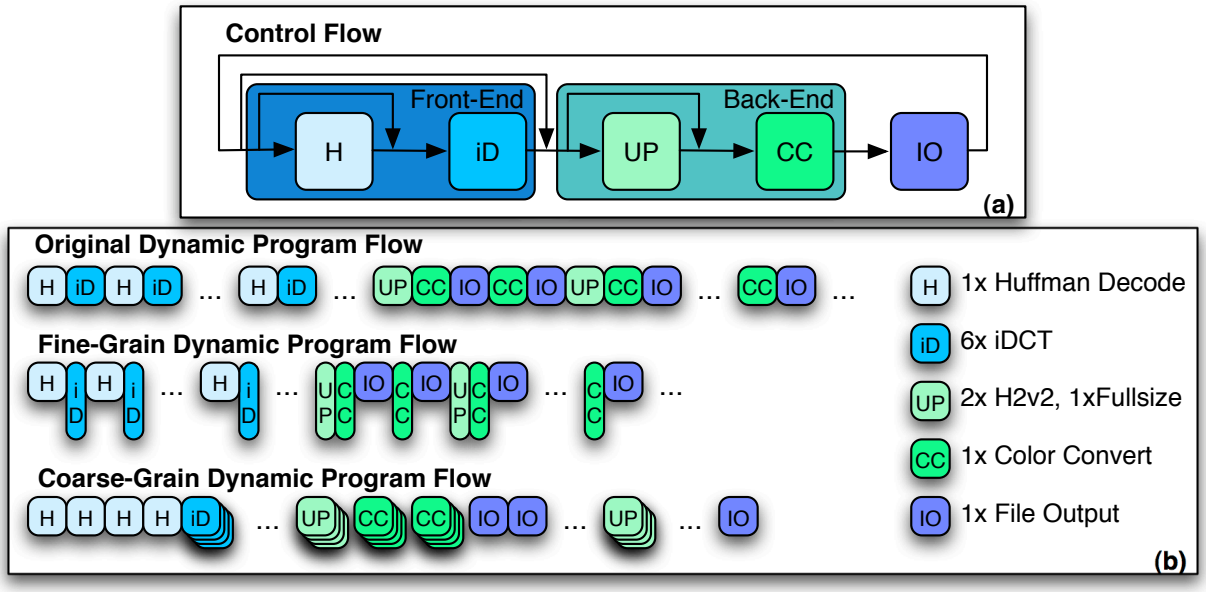


Figure 4.2 *jpegdec* High-Level Flows and Select Parallel Opportunities

Figure 4.2(a) shows the high-level control flow of the benchmark and the dynamic program flow (b) for the original implementation, fine-grain parallel implementation, and a coarse-grain implementation. The parallel implementations will be discussed in more detail in Sections 4.1.2 and 4.1.4. Each block in the lower portion of the diagram corresponds to a component of the decoding algorithm. To simplify the picture, components that are executed in inner loops have been merged into a single block and labeled (in the legend) with the iteration count of the inner loop. For example, in this configuration, iDCT is always called in groups of 6.

The front- and back-end processing are illustrated at a high level in Figures 4.3 and 4.4, respectively. The left sides show the different data representations, while the right sides show the control flow for the processing. These are intended not to illustrate exactly how data is maintained in memory, but to provide an algorithmic view. The different data representations are labeled on the left, and these labels are shown on the arcs in the control flow graphs to illustrate the inputs and outputs of the different stages of processing.

4.1.1 JPEG terminology

A *sample* is a single data point (generally one byte) for a single color component. A group of samples (in the case of this study, three samples) make up a *pixel*, which contains all of the information for that portion of an image. Rows and columns of pixels make up the resultant

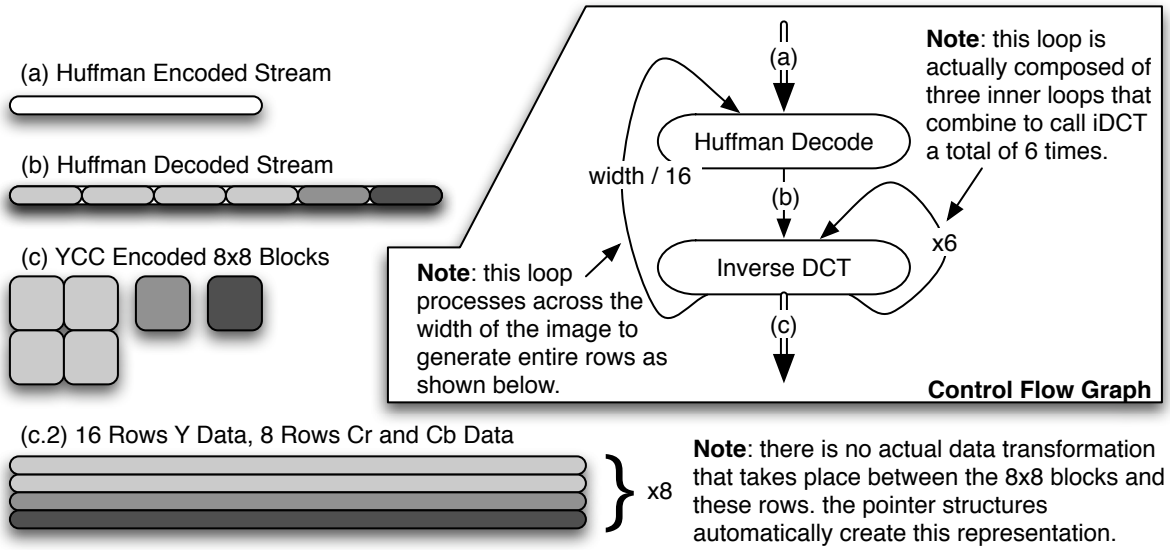


Figure 4.3 *jpegdec* Front-End High-Level Algorithm

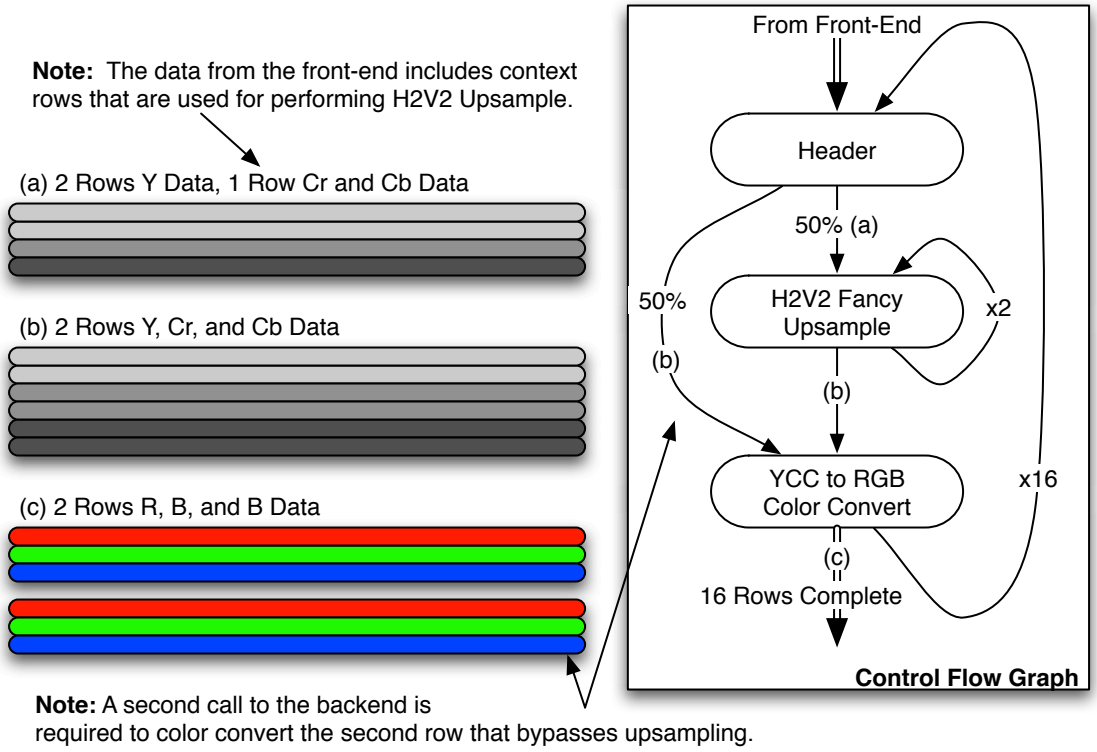


Figure 4.4 *jpegdec* Back-End High-Level Algorithm

image. Images are encoded into different color *components*. In this study, we will focus on the YCC and RGB color component schemes, but others do exist and are supported by *jpegdec*. The RGB scheme contains red, green, and blue components. It is commonly used as the output color scheme because RGB is used to display images on monitors. YCC is a color space made up of luminance (Y) and red and blue chrominance (Cr, Cb) samples, and is commonly used by JPEG encoding because the chrominance components can be downsampled without loss of perceptible image quality. These components tend to vary less than Y, and the human eye is also less sensitive to them. *Downsampling* refers to the lossy process of shrinking the uncompressed data set by averaging groups of samples into smaller sets (this is performed in *jpegenc*). *Upsampling* is the reverse process that is performed in *jpegdec*. In this study, H2V2 Fancy Upsample is used on the Cr and Cb components (no upsampling is necessary on the Y component). It takes the downsampled component frames and doubles their size in both the horizontal and vertical dimensions. It is "fancy" because it does not simply take each input sample and replicate it into three additional copies, but generates the new samples by taking a weighted average of surrounding downsampled samples.

An *MCU row* corresponds to a *Minimum Coded Unit* of data, which consists of the data necessary to generate a group of rows in the output image. When images are encoded in the JPEG format, DCT is performed separately on 8x8 blocks of samples. Groups of these 8x8 blocks for the different color components that correspond to a region in the original uncompressed image make up an *MCU block*. This study focuses on the 4:1:1 YCC compression scheme, in which for every four blocks of 8x8 Y samples, there exists one block of Cr and one of Cb. In this case, each MCU block is made up of 4 Y, 1 Cr, and 1 Cb block. An MCU row is a series of these blocks that stretch across the width of an image. Upsampling is performed on the Cr and Cb blocks so that there are equal numbers of Y, Cr, and Cb samples. These groups of three samples can then be color converted into RGB pixels.

This thesis will refer to the functions called from decompress_onepass() as the *front-end* and those called from sep_upsample() as the *back-end* processing (Figure 4.1). Unlike the terminology presented in the previous paragraphs, this is not standard terminology. The front-end performs Huffman decoding and iDCT, and then passes rows of YCC data to the back-end for upsampling and conversion to the RGB color scheme. The front-end processes 8x8 blocks of samples while the back-end processes rows of samples.

Due to the length of this section, the *jpegdec* evaluation is broken into separate sections on the front- and back-ends. Each section will provide an overview of the different granularities of parallelism followed by an evaluation of the necessary analysis and transformations.

4.1.2 Front-end parallelism

The front-end of *jpegdec* performs Huffman Decoding and iDCT. Huffman Decoding is inherently sequential. Without modifications to the encoding scheme, it is not possible to extract any meaningful parallelism from this task, and it is the critical recurrence in the application that is the fundamental limiting factor to the overall performance benefit that one can extract through parallel transformations. Each iDCT that is performed on each component of an MCU block is completely independent from all other blocks and all other components. As such, there are a variety of options for where to parallelize this stage.

4.1.2.1 iDCT - Fine-grain parallelism

First, we will examine the iDCT code for fine-grain parallelism. Figure 4.5 shows a selection of the iDCT code (many of the 200 lines have been removed for simplicity). There are two phases to an iDCT calculation (the loops at lines 171 and 276). The first phase generates a "workspace" of data from the input buffer, which is then processes by the second phase. Each iteration of phase 1 generates a column of the workspace, while each iteration of phase 2 processes a row from this workspace. Therefore, it is necessary to calculate the entire workspace before performing phase 2, and loop fusion is not possible. However, both of the two phase loops are entirely parallel. DCTSIZE is defined as 8, and therefore it is possible to spawn eight independent threads from the different loops iterations. In both phases, there exists a special case for each iteration that checks if the input data is all zero, and continue is used to skip the computation in this case. It is not uncommon for this case to be used, and therefore it may be useful to spawn threads after performing this check.

4.1.2.2 iDCT - Block-level parallelism

Having explored the fine-grain parallelism available in iDCT, we will evaluate more coarse-grain opportunities one function up in the call stack. Figure 4.6 presents the code for decompress_onepass(), which calls both Huffman Decoding and iDCT. Figure 4.7(a) provides a high-level view of the initial front-end sequential implementation. The two outer loops at lines 160 and 162 have been merged into a single outer loop in this figure for simplicity. In the current runtime configuration, the outer loop (line 160) always iterates exactly one time.[2]

[2]This study focusses on noninterleaved images. This loop iterates over MCU rows that make up an *iMCU row* in interleaved images, and can iterate as many as four times in those images.

```
147 GLOBAL(void)
148 jpeg_idct_islow (j_decompress_ptr cinfo, jpeg_component_info * compptr,
149                  JCOEFPTR coef_block,
150                  JSAMPARRAY output_buf, JDIMENSION output_col)
151 {
...
161   int workspace[DCTSIZE2];      /* buffers data between passes */
163
164   /* Pass 1: process columns from input, store into work array. */
...
168   inptr = coef_block;
170   wsptr = workspace;
171   for (ctr = DCTSIZE; ctr > 0; ctr--) {
...
...      ==== COMPUTATION HIDDEN FOR SIMPLICITY ===
257     wsptr[DCTSIZE*0] = (int) DESCALE(tmp10 + tmp3, CONST_BITS-PASS1_BITS);
258     wsptr[DCTSIZE*7] = (int) DESCALE(tmp10 - tmp3, CONST_BITS-PASS1_BITS);
259     wsptr[DCTSIZE*1] = (int) DESCALE(tmp11 + tmp2, CONST_BITS-PASS1_BITS);
260     wsptr[DCTSIZE*6] = (int) DESCALE(tmp11 - tmp2, CONST_BITS-PASS1_BITS);
261     wsptr[DCTSIZE*2] = (int) DESCALE(tmp12 + tmp1, CONST_BITS-PASS1_BITS);
262     wsptr[DCTSIZE*5] = (int) DESCALE(tmp12 - tmp1, CONST_BITS-PASS1_BITS);
263     wsptr[DCTSIZE*3] = (int) DESCALE(tmp13 + tmp0, CONST_BITS-PASS1_BITS);
264     wsptr[DCTSIZE*4] = (int) DESCALE(tmp13 - tmp0, CONST_BITS-PASS1_BITS);
...
268     wsptr++;
...
269   }
270
271   /* Pass 2: process rows from work array, store into output array. */
...
275   wsptr = workspace;
276   for (ctr = 0; ctr < DCTSIZE; ctr++) {
277     outptr = output_buf[ctr] + output_col;
...
287     if ((wsptr[1] | wsptr[2] | wsptr[3] | wsptr[4] | wsptr[5] | wsptr[6] |
288          wsptr[7]) == 0) {
...
...       ==== SET outptr[0 to 7] ====
301
302       wsptr += DCTSIZE;         /* advance pointer to next row */
303       continue;
304     }
...
...     ==== HEAVY CALCULATION - input: wsptr[0 to 7] ====
357     /* Final output stage: inputs are tmp10..tmp13, tmp0..tmp3 */
358
359     outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp3, ...
362     outptr[7] = range_limit[(int) DESCALE(tmp10 - tmp3, ...
365     outptr[1] = range_limit[(int) DESCALE(tmp11 + tmp2, ...
368     outptr[6] = range_limit[(int) DESCALE(tmp11 - tmp2, ...
371     outptr[2] = range_limit[(int) DESCALE(tmp12 + tmp1, ...
374     outptr[5] = range_limit[(int) DESCALE(tmp12 - tmp1, ...
377     outptr[3] = range_limit[(int) DESCALE(tmp13 + tmp0, ...
380     outptr[4] = range_limit[(int) DESCALE(tmp13 - tmp0, ...
...
384     wsptr += DCTSIZE;           /* advance pointer to next row */
385   }
386 }
```

Figure 4.5 *jpegdec* iDCT - jidctint.c:jpeg_idct_islow()

```
146 METHODDEF(int)
147 decompress_onepass (j_decompress_ptr cinfo, JSAMPIMAGE output_buf)
148 {
...
159   /* Loop to process as much as one whole iMCU row */
160   for (yoffset = coef->MCU_vert_offset; yoffset < coef->MCU_rows_per_iMCU_row;
161        yoffset++) {
162     for (MCU_col_num = coef->MCU_ctr; MCU_col_num <= last_MCU_col;
163          MCU_col_num++) {
164       /* Try to fetch an MCU.  Entropy decoder expects buffer to be zeroed. */
165       jzero_far((void FAR *) coef->MCU_buffer[0],
166                 (size_t) (cinfo->blocks_in_MCU * SIZEOF(JBLOCK)));
167       if (! (*cinfo->entropy->decode_mcu) (cinfo, coef->MCU_buffer)) {
168         /* Suspension forced; update state counters and exit */
169         coef->MCU_vert_offset = yoffset;
170         coef->MCU_ctr = MCU_col_num;
171         return JPEG_SUSPENDED;
172       }
...
178       blkn = 0;                 /* index of current DCT block within MCU */
179       for (ci = 0; ci < cinfo->comps_in_scan; ci++) {
180         compptr = cinfo->cur_comp_info[ci];
181         /* Don't bother to IDCT an uninteresting component. */
182         if (! compptr->component_needed) {
183           blkn += compptr->MCU_blocks;
184           continue;
185         }
186         inverse_DCT = cinfo->idct->inverse_DCT[compptr->component_index];
187         useful_width = (MCU_col_num < last_MCU_col) ? compptr->MCU_width
188                                                     : compptr->last_col_width;
189         output_ptr = output_buf[ci] + yoffset * compptr->DCT_scaled_size;
190         start_col = MCU_col_num * compptr->MCU_sample_width;
191         for (yindex = 0; yindex < compptr->MCU_height; yindex++) {
192           if (cinfo->input_iMCU_row < last_iMCU_row ||
193               yoffset+yindex < compptr->last_row_height) {
194             output_col = start_col;
195             for (xindex = 0; xindex < useful_width; xindex++) {
196               (*inverse_DCT) (cinfo, compptr,
197                               (JCOEFPTR) coef->MCU_buffer[blkn+xindex],
198                               output_ptr, output_col);
199               output_col += compptr->DCT_scaled_size;
200             }
201           }
202           blkn += compptr->MCU_width;
203           output_ptr += compptr->DCT_scaled_size;
204         }
205       }
206     }
207     /* Completed an MCU row, but perhaps not an iMCU row */
208     coef->MCU_ctr = 0;
209   }
...
219 }
```

Figure 4.6 *jpegdec* Front-End - jdcoefct.c:decompress_onepass()

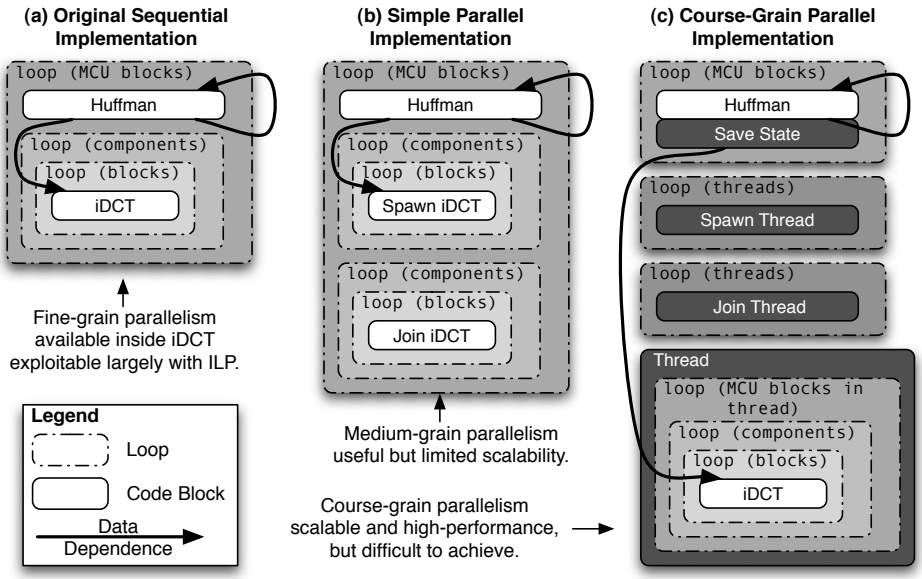


Figure 4.7 *jpegdec* Front-End Parallelism

The first simple technique (Figure 4.7(b)) for parallelizing this block of code performs each iDCT call in a separate thread. Because each MCU block contains six different blocks (4 Y, 1 Cr, 1 Cb), this will create six independent threads.

4.1.2.3 Full front-end course-grain parallelism

In order to achieve more scalable and coarser-grain parallelism, it is necessary to perform multiple iDCTs for multiple MCU blocks in each thread. One technique for achieving coarse-grain parallelism spawns one thread for each MCU block. Another technique (Figure 4.7(c)) performs loop distribution and converts the outer loop into two separate loops, one to perform the sequential Huffman decoding and one the parallel iDCT. This implementation provides better control over the number of threads and scales the computation size accordingly. For example, if 60 MCU blocks were to be processed, 5 threads would process 12 blocks each while 10 threads would process 6 blocks each.

4.1.3 Front-end transformations and analysis

Having provided an overview of the different opportunities for parallel execution of the front-end of *jpegdec*, this section will detail some of the analysis and transformation requirements for extracting the different parallel implementations. It will start with the fine-grain opportunities and build up to the more complicated coarse-grain versions.

4.1.3.1 iDCT analysis - Fine-grain analysis and transformation

Parallelizing the two inner phase loops of iDCT is the finest-granularity of thread-level parallelism explored here. This only requires intraprocedural analysis and trivial transformations, and, to the best of our knowledge, will be handled by the existing infrastructure once limitations in FULCRA with allocation pools and multidimensional pointer arrays are addressed (Section 5.1.2).

In phase 1 of iDCT, it is necessary to determine that there are no true scalar dependences around the loop-back edge for the various temporary variables. This is necessary because the temporary variables are declared at the top of the function rather than inside the loop scope (like they should be). The existing SSA can remove these loop-carried dependences. Once this is complete, the only other requirement is that the sections of the workspace that are modified in each iteration are disjoint. Each iteration modifies wsptr[DCTSIZE*(0 to 7)], and after each iteration wsptr is incremented. In other words, the first iteration modifies indices 0, 8, 16, ..., the second 1, 9, 17, ..., and so on. Equation (4.1) shows the induction summary for each iteration for workspace where ctr is the induction variable.

$$workspace[ctr + 8 * (0\ to\ 7)] \tag{4.1}$$

Because the loop iterates eight times, and the dependence distance can be shown to be eight, there are no loop-carried dependences. Basic Omega Test can determine this because the for loop has a compile-time static constant range and simple induction expressions.

The analysis of phase 2 is similar to that of phase 1, but is complicated by the fact that the memory that it writes to is based on the output_buf JSAMPARRAY. For now, let us assume that this array has the same semantics as a two dimensional array (this will be explored in greater detail in Section 4.1.3.2). With this assumption, the existing Omega Test should create the induction expression shown in Equation (4.2) and determine that the iterations are independent.

$$output\_buf[ctr][0\ to\ 7] \tag{4.2}$$

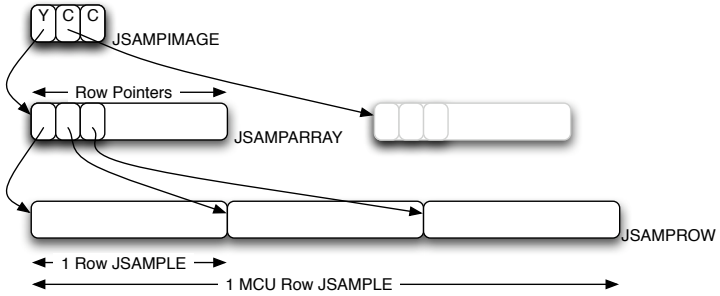


Figure 4.8 *jpegdec* Front-End Output Buffer Memory Layout

There are no major transformations that are necessary to parallelize these inner loops. Turning loops into thread spawn points has already been implemented in IMPACT. If it was not possible to perform the analysis to prove that the data written by phase 2 never overlapped, one could insert test code that determined whether it would be legal to assume that those components were all independent.

4.1.3.2 Output buffer

In general, when parallelizing loops, it is necessary to perform analysis to understand the nature of the data that each iteration of the loop writes. For each level of parallelism expressed in Section 4.1.2, it is necessary for an analysis to disambiguate references to the output buffer. Therefore, we will first provide an overview of how the output buffer is maintained. Then we will evaluate how an analysis would identify the independence of writes to this structure.

Figure 4.8 presents a high-level picture of how the front-end's output buffer is maintained in memory. It is essentially a three-dimensional array allocated as pointers in memory. The row buffers at the bottom of the figure are shown as a single contiguous memory space because they are allocated in that manner (Section 4.1.6). Moving forward, we will assume that the output buffer structure maintains the semantics of a multidimensional array, although the current infrastructure does not have the ability to determine this.

Figure 4.9 shows a more detailed picture of how the output buffer is maintained. In order to handle the need for context data in upsampling,[3] and to remove the need for memcpy calls, two separate sets of row pointer arrays are maintained. Every time the front-end processes a single MCU row, it switches between one of the two possible xbuffer sets. This complicates analysis by making it difficult to generate an accurate set of dependence information for

[3] H2V2 fancy upsampling requires context data from the surrounding samples.

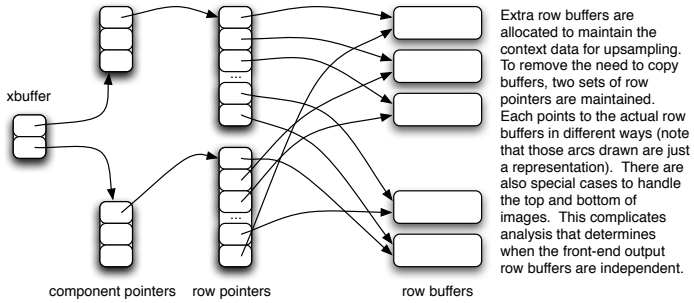


Figure 4.9 *jpegdec* Front-End xbuffer

the different output row buffers. While the row pointers within the same invocation of decompress_onepass() will always point to disjoint arrays, this is not the case between different invocations. For example, row pointer r from iteration i could be the same as row pointer $r+1$ for iteration $i+1$.

Now that we have an understanding of how the output buffers are maintained in memory, we will evaluate what needs to be done to disambiguate different output buffer accesses. It is generally necessary to start at the inner loops and generate summaries of the accesses while moving outwards. Therefore, the first step is to evaluate the loops in iDCT.

The phase 1 loop (jidctint.c:171) modifies data on the local stack, and therefore can be ignored. The phase 2 loop (jidctint.c:276) modifies data from the output_buf pointer parameter. The output location is determined at line 277. For each iteration of the loop (line 276), a row pointer is selected from the output_buf two dimensional array. Once this is complete, output_col is used to jump to the correct position inside of the row array. From this point, 8 samples are written. Therefore, each iteration modifies 8 samples from the same column in 8 different rows. If it were possible to determine that the output_buf structure always maintained array semantics, and that each row pointer was distinct and pointed to disjoint objects, then one could prove that there was no loop-carried output-dependence due to output_buf in iDCT. To summarize, each iDCT call modifies:

$$output\_buf[0\ to\ 7][output\_col\ to\ output\_col+7] \tag{4.3}$$

Before investigating the characteristics of decompress_onepass(), it is important to note that the output_buf in this context is a three dimensional array (JSAMPIMAGE). This is different

from iDCT, where it is a two-dimensional array (JSAMPARRAY). decompress_onepass() calculates output_ptr and output_col, which are passed into iDCT. output_ptr is passed into the output_buf iDCT parameter.

Having summarized iDCT, it is now possible to evaluate the characteristics of the loops in decompress_onepass() that iterate through the 8x8 blocks in a single component of an MCU block. These loops are active for the Y component, which contains four blocks in each MCU block. Line 189 sets the initial output_ptr. Value flow information can show that yoffset is 0. This sets the base address to output_buf[ci]. The output_ptr is then incremented by DCT_scaled_size on line 203 for every iteration through the yindex loop at line 191. Value flow shows that this value is 8. Combining these results in the following induction expression:

$$output\_ptr = output\_buf[ci] + 8 * yindex \tag{4.4}$$

The output_col also determines which section of output_buf is modified by each iDCT call (as shown in Equation (4.3)). Using value flow information for DCT_scaled_size, along with lines 194 an 199, it can be shown to be:

$$output\_col = start\_col + 8 * xindex \tag{4.5}$$

Equations (4.4) and (4.5) can be substituted into (4.3) to prove that the iDCT calls in this context access disjoint regions of output_buf. Each call modifies:

$$output\_buf[ci][8 * yindex + (0\ to\ 7)][start\_col + 8 * xindex + (0\ to\ 7)] \tag{4.6}$$

Having summarized the region of output_buf that is modified in the loops that iterate through the blocks for a single component, we can now examine the loop at 179 that iterates through the three image components. This context does not contain any additional information, except that ci iterates from 0 to comps_in_scan-1. Value flow will show that this from 0 to 2.

In context of the outer loops at lines 160 and 162, an induction expression for start_col is necessary. Its induction expression is exclusively based on the calculation at line 190. In order for the regions of output_buf to be disjoint at this scope, it must be possible to determine:

$$MCU\_sample\_width \geq 8 * MCU\_width \tag{4.7}$$

Value-flow can be used to verify the relationship described above is always true. This will show that each iDCT call inside each decompress_onepass() call will modify disjoint regions of output_buf.

4.1.3.3 iDCT - Block level analysis and transformation

Having evaluated the characteristics of output_buf at different loop nests, we can now investigate more coarse-grain parallel implementations than the one presented in Section 4.1.3.1. Beyond the inner-loop parallelism within iDCT, one can perform the six iDCT calls in parallel based on the loops at jdsample.c:179 and 191 (Figure 4.7(b)). Note that for simplicity the two inner loops in the figure have been merged into a single loop in the figure. The only necessary transformation here is the addition of two new loops to perform the thread join and convert the iDCT calls into thread spawn calls. This transformation will not require any modification inside of the iDCT function. The only analysis that is necessary to parallelize at this scope is to prove that the sections of output_buf are disjoint (Section 4.1.3.2).

IMPACT currently contains a transformation that allows threads to be spawned off of an inner loop. In order to extract parallelism at this scope, one would need to extend this existing transformation to handle multiple nested loops.

4.1.3.4 Full front-end course-grain analysis and transformation

The final level of parallelism creates threads for multiple calls to iDCT across multiple MCU blocks through the loop at jdcoefct.c:162. This is more complicated than the prior techniques, but also provides more scalable and coarse-grain parallelism. Loop distribution is required to separate Huffman decoding and iDCT into separate loops. Figure 4.7(b) illustrates this transformation, which not only requires loop distribution to handle multiple loop nests, but also a side exit (the return at line 171). While profiling will show that this side-exit is never triggered, it must still be handled in the transformation. Array expansion [25] is also necessary to perform this distribution because there exists array dataflow from Huffman to iDCT. It will be necessary to store the different MCU_buffer states after each decoding, and then process them in the new iDCT loops.

Additional analysis beyond those described in Section 4.1.3.2 will be necessary for attaining this level of parallelism. By analyzing decompress_onepass(), it is not possible to determine whether there is a loop-carried dependence around the loop at line 162 for MCU_buffer. The Huffman decoding function is quite complicated, and generating a realistic summary of it

would be quite difficult. However, jzero_far() (line 165) initializes the MCU_buffer to zero before Huffman decoding. By proving that this function initializes all data that is used by iDCT, there are no true loop-carried dependences due to MCU_buffer. It overwrites blocks_in_MCU * 64 shorts of data (sizeof(JBLOCK) is the size of a 64 elemenent array of shorts). Now it must be shown that this covers the data read by iDCT.

An analysis of iDCT shows that 64 elements of data after the pointer parameter coef_block are used by phase 1 of the processing. MCU_buffer[blkn+xindex] is passed to this parameter, which can be simplified to MCU_buffer[(0 to 5)] using linear equations (this is nontrivial, and will be left as an excersie to the reader). Because blocks_in_MCU is six, the data that is used by the iDCT calls is covered by the jzero_far call, and therefore the MCU_buffer data that exists from previous iterations is not relevant.

4.1.4 Back-end parallelism

As shown Figure 4.4, the back-end processes rows of data (unlike iDCT which works with 8x8 blocks). The back-end begins with 8 rows of downsampled data for the Cr and Cb components, and 16 rows for the Y component (which was not downsampled during encoding).[4] Upsampling is performed on the Cr and Cb components, one row at a time to generate two complete rows of data. This data is then combined with the Y row data and converted into the RGB color space 1 row at a time. The upsampling and color conversion routines have functionality to process multiple input rows for a single invocation, but are always used to process exactly one row of data in this configuration. Each call to sep_upsample() (the function that calls both h2v2_fancy_upsample() and ycc_rgb_convert()) will generate exactly one row of the output image. Figure 4.2(b) shows that upsampling is performed for every other color conversion. It is necessary to skip upsampling every other call because color conversion must process the second upsampled row. Figure 4.10 presents the high-level function that performs the back-end processing. The fine-grain parallelism will be explored first, followed by the more scalable coarse-grain parallelism.

4.1.4.1 Fine-grain upsample parallelism

The h2v2_fancy_upsample() function is implemented as a series of three nested loops, and is called indirectly from sep_upsample(). Figures 4.11 and 4.12 present the code and a diagram

[4] Different encoding schemes result in different row counts for each component. For example, if no downsampling was performed by the encoding process, there would be eight rows of Y, Cr, and Cb data.

```
 88 METHODDEF(void)
 89 sep_upsample (j_decompress_ptr cinfo,
 90               JSAMPIMAGE input_buf, JDIMENSION *in_row_group_ctr,
 91               JDIMENSION in_row_groups_avail,
 92               JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
 93               JDIMENSION out_rows_avail)
 94 {
 95   my_upsample_ptr upsample = (my_upsample_ptr) cinfo->upsample;
 96   int ci;
 97   jpeg_component_info * compptr;
 98   JDIMENSION num_rows;
 99
100   /* Fill the conversion buffer, if it's empty */
101   if (upsample->next_row_out >= cinfo->max_v_samp_factor) {
102     for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
103          ci++, compptr++) {
104       /* Invoke per-component upsample method.  Notice we pass a POINTER
105        * to color_buf[ci], so that fullsize_upsample can change it.
106        */
107       (*upsample->methods[ci]) (cinfo, compptr,
108         input_buf[ci] + (*in_row_group_ctr * upsample->rowgroup_height[ci]),
109         upsample->color_buf + ci);
110     }
111     upsample->next_row_out = 0;
112   }
113
114   /* Color-convert and emit rows */
115
116   /* How many we have in the buffer: */
117   num_rows = (JDIMENSION) (cinfo->max_v_samp_factor - upsample->next_row_out);
118   /* Not more than the distance to the end of the image.  Need this test
119    * in case the image height is not a multiple of max_v_samp_factor:
120    */
121   if (num_rows > upsample->rows_to_go)
122     num_rows = upsample->rows_to_go;
123   /* And not more than what the client can accept: */
124   out_rows_avail -= *out_row_ctr;
125   if (num_rows > out_rows_avail)
126     num_rows = out_rows_avail;
127
128   (*cinfo->cconvert->color_convert) (cinfo, upsample->color_buf,
129                                      (JDIMENSION) upsample->next_row_out,
130                                      output_buf + *out_row_ctr,
131                                      (int) num_rows);
132
133   /* Adjust counts */
134   *out_row_ctr += num_rows;
135   upsample->rows_to_go -= num_rows;
136   upsample->next_row_out += num_rows;
137   /* When the buffer is emptied, declare this input row group consumed */
138   if (upsample->next_row_out >= cinfo->max_v_samp_factor)
139     (*in_row_group_ctr)++;
140 }
```

Figure 4.10 *jpegdec* Backend - jdsample.c:sep_upsample()

```
344 METHODDEF(void)
345 h2v2_fancy_upsample (j_decompress_ptr cinfo, jpeg_component_info * compptr,
346                      JSAMPARRAY input_data, JSAMPARRAY * output_data_ptr)
347 {
348   JSAMPARRAY output_data = *output_data_ptr;
349   register JSAMPROW inptr0, inptr1, outptr;
350 #if BITS_IN_JSAMPLE == 8
351   register int thiscolsum, lastcolsum, nextcolsum;
352 #else
353   register INT32 thiscolsum, lastcolsum, nextcolsum;
354 #endif
355   register JDIMENSION colctr;
356   int inrow, outrow, v;
357
358   inrow = outrow = 0;
359   while (outrow < cinfo->max_v_samp_factor) {
360     for (v = 0; v < 2; v++) {
361       /* inptr0 points to nearest input row, inptr1 points to next nearest */
362       inptr0 = input_data[inrow];
363       if (v == 0)               /* next nearest is row above */
364         inptr1 = input_data[inrow-1];
365       else                      /* next nearest is row below */
366         inptr1 = input_data[inrow+1];
367       outptr = output_data[outrow++];
368
369       /* Special case for first column */
370       thiscolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
371       nextcolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
372       *outptr++ = (JSAMPLE) ((thiscolsum * 4 + 8) >> 4);
373       *outptr++ = (JSAMPLE) ((thiscolsum * 3 + nextcolsum + 7) >> 4);
374       lastcolsum = thiscolsum; thiscolsum = nextcolsum;
375
376       for (colctr = compptr->downsampled_width - 2; colctr > 0; colctr--) {
377         /* General case: 3/4 * nearer pixel + 1/4 * further pixel in each */
378         /* dimension, thus 9/16, 3/16, 3/16, 1/16 overall */
379         nextcolsum = GETJSAMPLE(*inptr0++) * 3 + GETJSAMPLE(*inptr1++);
380         *outptr++ = (JSAMPLE) ((thiscolsum * 3 + lastcolsum + 8) >> 4);
381         *outptr++ = (JSAMPLE) ((thiscolsum * 3 + nextcolsum + 7) >> 4);
382         lastcolsum = thiscolsum; thiscolsum = nextcolsum;
383       }
384
385       /* Special case for last column */
386       *outptr++ = (JSAMPLE) ((thiscolsum * 3 + lastcolsum + 8) >> 4);
387       *outptr++ = (JSAMPLE) ((thiscolsum * 4 + 7) >> 4);
388     }
389     inrow++;
390   }
391 }
```

Figure 4.11 *jpegdec* Upsampling - jdsample.c:h2v2_fancy_upsample()

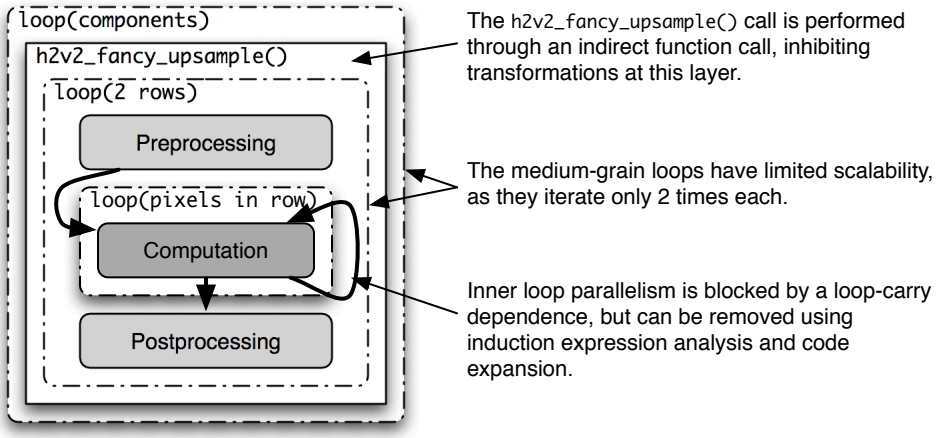


Figure 4.12 *jpegdec*:h2v2_fancy_upsample() - Sequential Implementaion

for this function, respectively. In the diagram, the outermost loop in h2v2_fancy_upsample() has been removed for simplicity because it only iterates a single time. A single component of the image is processed by each h2v2_fancy_upsample() call, and a fourth loop loop (the outermost loop in the diagram) in sep_upsample() (line 102 in Figure 4.10) iterates through the components. The middle loop (line 360) iterates two times (as shown in the diagram), once for each of the two resultant rows that will be generated from the upsample. The inner loop (line 376) iterates across all of the samples in an input row, with the exception of the first and last which are handled outside of the loop.

In h2v2_fancy_upsample(), the calculations necessary to upsample a pixel can be reused when upsampling surround pixels. For each iteration of the inner loop, two output samples in a row are generated. The values of these samples are generated from three calculations that are based on the input samples – lastcolsum, thiscolsum, and nextcolsum. For each iteration of the inner loop, a new value for nextcolsum is calculated. At the end of each inner loop iteration, thiscolsum is moved to lastcolsum and nextcolsum into thiscolsum, and a new nextcolsum is calculated at the beginning of each loop iteration. This creates a loop-carried dependence that must be removed in order to extract inner-loop thread-level parallelism. It is important to not that the loop-carried dependence need not be broken between every iteration, but only iterations that will be executed in different threads.

h2v2_fancy_upsample() generates two rows per invocation. These are generated by the loop at line 360. A simple but limited medium-grain parallelism could be extracted by performing the two iterations of this loop in parallel. The fourth outer loop (in sep_upsample()) calls

```
119 METHODDEF(void)
120 ycc_rgb_convert (j_decompress_ptr cinfo,
121                   JSAMPIMAGE input_buf, JDIMENSION input_row,
122                   JSAMPARRAY output_buf, int num_rows)
123 {
...
138   while (--num_rows >= 0) {
139     inptr0 = input_buf[0][input_row];
140     inptr1 = input_buf[1][input_row];
141     inptr2 = input_buf[2][input_row];
142     input_row++;
143     outptr = *output_buf++;
144     for (col = 0; col < num_cols; col++) {
145       y  = GETJSAMPLE(inptr0[col]);
146       cb = GETJSAMPLE(inptr1[col]);
147       cr = GETJSAMPLE(inptr2[col]);
148       /* Range-limiting is essential due to noise introduced by DCT losses. */
149       outptr[RGB_RED] =   range_limit[y + Crrtab[cr]];
150       outptr[RGB_GREEN] = range_limit[y +
151                               ((int) RIGHT_SHIFT(Cbgtab[cb] + Crgtab[cr],
152                                                  SCALEBITS))];
153       outptr[RGB_BLUE] =  range_limit[y + Cbbtab[cb]];
154       outptr += RGB_PIXELSIZE;
155     }
156   }
157 }
```

Figure 4.13 *jpegdec* Color Convert - jdcolor.c:ycc_rgb_convert()

h2v2_fancy_upsample() twice (once for both the Cr and Cb components). Using these two separate loops, four medium-grain parallel threads can be extracted.

4.1.4.2 Fine-grain color convert parallelism

Figure 4.13 presents the code for ycc_rgb_convert(), which performs color conversion from the YCC to RGB color scheme for num_rows rows. In the configuration studied, this is always a single row, and the outer loop (line 138) only iterates one time. The inner loop (line 144) iterates through all the YCC pixels in a row and converts them to RGB pixels. To extract fine-grain parallelism, one must create threads from this inner loop. It is possible to extract separate threads for each RGB component, but this would likely result in poor cache performance in a coherent shared memory system. Loop blocking, on the other hand, would perform well.

4.1.4.3 Course-grain back-end parallelism

The back-end also contains coarse-grain parallelism. Each row that is upsampled and color converted is independent of previous rows. At an algorithmic level, this is not difficult, but analyzing and transforming the existing implementation is nontrivial.

The control- and data-flow in the back-end of *jpegdec* is more complicated than the front-end. Figure 4.4 presents the high-level data formats and control flow, and Figure 4.10 presents code from sep_upsample(). Recall that because upsampling generates two rows of data and color conversion processes only a single row at a time. As a result, upsampling is only performed in every other call to sep_upsample(). Every call to sep_upsample() generates a single row of RGB data which is then written to file after returning up the call stack. While it might be possible to take advantage of the pattern illustrated in the bottom of Figure 4.2, the identification would be difficult and a more general solution is desirable. Also, the pattern is not the same for all YCC encoding schemes.

Because the *jpegdec* back-end does not contain any true data recurrences, it is possible to reorder the execution of different blocks of independent processing. As stated above, upsampling is performed followed by two data dependent calls to color convert. Because each row's upsampling calls are independent, one can achieve coarse-grain parallelism by delaying execution of the color convert (and the successive file output), and performing a series of upsamples for different rows in parallel.

The same technique can be applied to ycc_rgb_convert() and file output. Because the sizes and locations of the file writes are deterministic, it would even be possible to perform the file accesses in parallel. The performance benefit of this would be limited in this case – but in high-performance systems utilizing RAID disk systems performance improvements could be extracted.

4.1.5 Back-end transformation and analysis

4.1.5.1 Fine-grain upsample analysis and transformation

There are two different fine-grain parallel implementations of h2v2_fancy_upsample(). The first extracts parallelism from the inner loop (jdsample.c:376) while the second extracts limited parallelism from the two outer loop contexts (jdsample.c:360 (h2v2_fancy_upsample()) and jdsample.c:102 (sep_upsample())).

The inner loop can be parallelized by removing the loop-carried dependence for the colsum scalars, and extracting threads as presented in Section 4.1.4.1. To perform such a transformation, one must first identify the scalar loop-carried dependences and the induction expressions for generating those scalars. Once the expressions are identified, code expansion can be used to remove the loop-carried dependence between iterations in different threads. For each thread

that is spawned, initial values (that were originally defined around the back-edge) can be computed prior to entering the inner loop.

In the inner loop, there are four iterators – colcrtr (the loop counter) as well as *inptr0, *inptr1, and *outptr. Induction expressions can be calculated for each array access based on these pointers. The input pointer induction expressions may be useful for generating the initial colsum values for each thread. The output pointer induction expression proves that successive iterations are independent. These expressions can be calculated with the existing Lcode induction expression analysis.

It is also necessary to prove that the objects pointed to by outptr cannot intersect with those pointed to by inptr. Because these objects are allocated using a pool allocation[5] routine, Pcode and FULCRA cannot disambiguate the objects. A future research topic may be the identification of allocation pools, but in the near-term pragmas can be used to identify these routines.[6] This problem exists throughout the functions in *jpegdec*, and it will be necessary to add this functionality to IMPACT to extract any level of parallelism from many of the functions presented here.

Another roadblock to disambiguating the outptr and inptr accesses is caused by the Y component's upsample function, fullsize_upsample(). Because Y does not need to be upsampled, this function creates outptr by simply copying inptr. This operation will merge the objects in FULCRA. Without new analysis, it may not be possible to prove that these objects are disjoint in the context of h2v2_fancy_upsample(). Code can be inserted to verify that the objects are disjoint when performing h2v2_fancy_upsample().

The second parallel implementation of h2v2_fancy_upsample() extracts four threads using the loops at jdsample.c:360 (Figure 4.11) and jdsample.c:102 (Figure 4.10). Interprocedural analysis and transformation will be necessary to perform this unless h2v2_fancy_upsample() is inlined in Pcode. Performing an interprocedural analysis around the indirect function call will not be covered in this section – it will be assumed that either the function has been inlined into sep_upsample() or that an interprocedural framework is developed that allows the same type of analysis to be performed. These two loops will need to be replicated – the original will spawn threads that perform the block of code between lines 361 and 387 in h2v2_fancy_upsample(), while the new loops will join those threads. In addition to proving that the input and output

[5]Memory allocation using malloc is generally inefficient, particularly when allocating many small pieces of data. Pool allocation uses malloc to allocate a large chunk of memory, and then provides pieces of this chunk as they are requested by the application, reducing the overhead of performing heap allocations.

[6]FULCRA contains support for marking different function calls as having the semantics of malloc, but this functionality is currently not utilized by IMPACT.

pointers access distinct objects, it is now necessary to prove that the different iterations modify different sections of the outptr data. In order to prove this, the output characteristics will be evaluated from the inner loops out.

For each upsampled row that is generated (based on the loop at jdsample.c:360), data is written to output_data[outrow++] (line 367). output_data is allocated as an array of pointers to arrays of a row of sample components (JSAMPLE). As in the front-end of *jpegdec*, it will be necessary to add support for analysis that can prove (or a transformation that can check) that such structures have the semantics of true multidimensional arrays. These arrays are allocated using the alloc_sarray subroutine (called from jinit_upsampler()) which allocates an array of pointers to arrays of data with as few malloc calls as possible (Section 4.1.6). This information is sufficient to prove that different iterations of loop 360 modify different output data. Assuming that the loop at line 359 iterates only a single time (based on profiling data), each call to h2v2_fancy_upsample() modifies the following:

$$output\_data[0\ to\ 1][0\ to\ downsampled\_width * 2] \tag{4.8}$$

In the next loop-nest out (sep_upsample() at jdsample.c:102), analysis must determine that the output data for different iterations is disjoint. upsample→color_buf + ci is passed into h2v2_fancy_upsample() where it is dereferenced at line 348 to get the output_data pointer in the paragraph above. This reduces to the following where ci ranges from 0 to 2 (using value-flow and profiling):

$$color\_buf[ci][0\ to\ 1][0\ to\ downsampled\_width * 2] \tag{4.9}$$

The values of the pointers in the component array are set only one time (jinit_upsampler()), so it should be possible to prove that they point to distinct objects, and therefore the different calls to h2v2_fancy_upsample() modify different sections of the output buffer inside of the scope of the loop at jdsample.c:102.

4.1.5.2 Fine-grain color convert analysis and transformation

The ycc_rgb_convert() function can be transformed by spawning threads to color convert different ranges of pixels in a row. Loop blocking can be performed to extract such an implementation, and is currently under development in IMPACT. Basic induction expression analysis shows that there are no loop-carried dependences for outptr. The scalars used are generated at

the beginning of each loop iteration and used at the end. The only complicated analysis involves proving that outptr does not overlap inptr0, inptr1, inptr2, or the tables used to simplify the calculation. Existing pointer analysis handles the tables, but currently, because both the input and output buffers are allocated with alloc_sarry, does not handle the input and output pointers. As in the analysis of h2v2_fancy_upsample(), it will be necessary for the compiler to know that the alloc_sarray has malloc semantics. Once this is complete, fine-grain parallel compilation of ycc_rgb_convert() will be possible.

4.1.5.3 Course-grain back-end analysis and transformation

In order to extract coarse-grain parallelism in the back-end of *jpegdec*, it is necessary to substantially reorder the execution as described in Section 4.1.4.3. The problem space for this particular transformation is very large as the outermost loop involved also contains the front-end execution. It therefore is necessary to consider all of the dependence information throughout both the front- and back-ends of the application. This thesis will not evaluate the necessary analysis for such a transformation. However, the concept of transforming code to delay execution of certain sections is an interesting and powerful one.

In order to delay execution of sections of code, it is necessary to maintain only the relevant system state. How this is performed largely affects the possible performance that one can gain from such a transformation. The delayed execution of ycc_rgb_convert() demonstrates this point. Every other time that sep_upsample() is called, upsampling is not performed, and the state of the upsample output buffer is not modified. As such, creating a copy of the upsample buffers for every skipped call of upsampling is not necessary.

Intelligently delaying execution is also important for reducing the overhead of this type of transformation. ycc_rgb_convert() uses data for the Y component directly from the front-end output. Therefore, delaying execution across instantiations of the front-end processing is possible if the front-end data is further buffered. This is possible, but would cause more substantial program transformations and greater analysis scope. An easier solution would detect situations when the application was going to enter potentially dangerous (or unanalyzable code), and drain the delayed execution at that point. For example, ycc_rgb_convert() invocations could be delayed until it was determined that the front-end was going to execute, and then be executed. Such techniques can help to reduce the strain on the analysis, and more importantly will help to enable this type of transformation in cases that otherwise would have been prevented.

```
1: procedure SIMPLE 2D ARRAY ALLOCATION
2:     RowArray ← malloc(NumRows)
3:     for all row in NumRows do
4:         RowArray[row] ← malloc(NumCols)
5:     end for
6: end procedure
```

```
1: procedure OPTIMIZED 2D ARRAY ALLOCATION
2:     RowArray ← malloc(NumRows)
3:     for all row in NumRows do
4:         if Need to Allocate then        ▷ On modern machines this generally fires one time
5:             size ← MIN(space_left, max_malloc_size)
6:             buffer ← malloc(size)
7:             count ← 0
8:         end if
9:         RowArray[row] ← buffer[count]
10:        count ← count + 1
11:    end for
12: end procedure
```

Figure 4.14 Two-Dimensional Array Allocation

4.1.6 Pointer multidimensional arrays

As shown in Figure 4.14, there are (at least) two different techniques for allocating two dimensional arrays as an array of pointers to arrays. The top "simple" algorithm first allocates an array for the pointers and then calls malloc for each row that is allocated. This function can be optimized by using an internal allocation pool. The bottom "optimized" algorithm allocates an array of pointers (like in the simple case), but calls malloc with a size larger than the size of a single row and then increments pointers into this buffer for successive rows. This is the technique that is used in alloc_sarray() in *jpegdec*. Both of these allocations result in a structure that has the same semantics as a standard array allocation in C assuming the pointer array is never modified.

4.1.7 *jpegdec* summary

The *jpegdec* benchmark contains many useful granularities of parallelism, and the variety of analysis and transformation techniques make it an excellent benchmark to target for both near- and long-term studies. The inner loops of the benchmark, including those in iDCT, up-sampling, and color conversion, are excellent metrics for near-term work. The coarse-grain

transformations in the front-end provide a good midterm goal, while the transformations in the back-end are more long-term. The analysis requirements build upon each other as the granularity of parallelism increases. This benchmark also motivates the many analysis challenges that we face for performing coarse-grain interprocedural work.

In *jpegdec*, fine-grain parallelism requires simple transformations, including trivial ones to allow nested loops to spawn threads, and slightly more complicated ones to remove loop-carried dependences (h2v2_fancy_upsample()). Transforming these loops should be a priority in near-term work. Not only are the transformations necessary, but the existing IMPACT analysis infrastructure is capable of removing the majority of the spurious dependences. Extending FULCRA to deal with allocation pools (through pragmas) will be necessary to remove some of the remaining spurious dependences at this granularity. This application also uses arrays of pointers to implement multidimensional arrays. Analysis or programmer pragmas will be necessary for the compiler to be able to understand that certain accesses have the semantics of multidimensional arrays.

Developing transformations to extract parallel execution in the front-end is another near-term target. There are a variety of levels of parallel execution available in the front-end, each of which has different requirements. Loop distribution that supports multiple loop nests with side exits will be necessary to extract the highest granularity of parallelism.

The front-end also motivates how the analysis performed at one level of granularity is leveraged for higher levels of parallelism. Similar to what is done currently in FULCRA, it will be useful to develop a framework that summarizes only the necessary information to higher levels of analysis – particularly for interprocedural cases. For example, loop-carried dependences inside of inner loops of a block may not be important, while information about the data spaces that are accessed and written to by these blocks may be.

Creation of array gen-kill data will be necessary for removing dependences in certain loops. This data commonly depends on execution-constant loop bounds that are stored in heap-allocated structures. The ability to identify these constants, or at least potential relationships between different constants, will be necessary to properly summarize the data that different loops access. Without such accurate summaries, performing certain disambiguations will not be possible. Value-flow information will not only be important for loop bounds, but is also necessary for simplifying other aspects of induction expressions.

Transforming the back-end for coarse-grain parallel execution requires a delayed execution model. While such a model will be applicable in a variety of different situations, creating such a generalized transformation in IMPACT will be quite challenging.

4.2 LAME - MP3 Encoder

LAME[7] is an open-source mp3 encoder application that incorporates various audio compression techniques. Each of these techniques exhibits different characteristics – both in output files and the parallelism that is inherent to their algorithms.

In general, mp3 encoding contains two major high-level steps. The first step takes a section, or *frame*, of uncompressed audio and performs an analysis on it to determine how it should be encoded. We will call this *noise analysis*. This process not only considers the audio that it is currently encoding, but also the results of past analysis. This is necessary to avoid decisions that would result in audible artifacts in the output stream, and therefore successive invocations cannot be performed in parallel. The second step of the encoding process is the compression, which uses the noise analysis calculations to determine how the frame should be encoded.

While the high-level mp3 encoding algorithm is relatively simple and constant, there are three different techniques for performing the encoding: Constant Bit Rate (CBR), Average Bit Rate (ABR), and Variable Bit Rate (VBR). CBR is generally the most common encoding technique despite the fact that it generally provides the worst ratio of audio quality to file size. In this mode, the user chooses a compression bitrate, and that is used throughout the encoding process. Other than having a predictable file size, there is no good reason to use this compression technique.[8] ABR is similar to CBR in that the user provides a bitrate to the application. Unlike CBR, this bitrate is not fixed for each frame, and more complicated portions of the audio stream can be encoded at a higher bitrate. VBR takes the concept of allowing variations in the bitrate one step farther; the encoder has full control over the bitrates for different sections of the audio stream. The user can specify a "quality setting," which is an integer from 0 to 9 where the larger integers result in larger and higher-quality files. VBR generally produces the highest ratio of audio quality to file size, and is the mode that is used in this study.

There are many encoding settings that the user can specify, but there is one specific setting that is important to this thesis. *LAME* incorporates a *bit reservoir* that allows for leftover space from prior frames to be used for frames that may need more accuracy. As we will show later, this feature is important because it has a strong effect on the parallelism inherent to the compression component of the encoding process. While each of the techniques can make use

[7] "LAME" is a recursive acronym that stands for **L**AME is not **A**n **M**P3 **E**ncoder.

[8] Traditionally, the other encoding techniques, VBR in particular, were less stable and occasionally resulted in poor audio quality. There has been extensive work on these encoding algorithms over the years, and this is no longer the case.

of this feature, the output quality of audio encoded with CBR can be heavily affected by the additional accuracy afforded by the bit reservoir. In ABR, disabling this feature may have some effect on the quality of the output audio, as some of the bitrates may have to be scaled back to maintain the average bitrate. In VBR, because the output file size is not considered in making compression decisions, it is only the size of the file that is affected by disabling this feature. Because VBR is able to pick from a variety of bitrates, this does not have a substantial effect on the output file size. We have seen increases on the order of 10%.

Figure 4.15 shows the pruned[9] callgraph for VBR *LAME*. The noise analysis and compression components of the code have been boxed out. From this picture, we can see that the majority of the runtime is spent performing compression. However, parallelizing this section of the code is not sufficient for attaining high performance. The sequential noise analysis section contributes to about one sixth of the runtime, and therefore Amdahl's law prevents performance benefits of more than 6x without targeting this code as well.

LAME requires coarse-grained parallelism in order to achieve a high-performance parallel implementation. Complicated interprocedural analysis and transformations are necessary in order for a compiler to attain such an implementation. There are some simpler and less scalable opportunities (encoding the left and right channels in parallel for example), and these will be evaluated in the context of noise analysis. Because of the complicated nature of *LAME*, this section will focus more on transformations and less on the analysis. Specific considerations for the interprocedural analysis framework will be motivated.

One important take-away from *LAME* is the concept of the critical recurrence. When parallelizing applications, it is not uncommon to encounter dataflow SCCs in loops that must be transformed out in order to leverage the parallel components. The SCC that exists through the most runtime intensive loop is termed the "critical recurrence." Transformations and analysis can be broken into two separate categories – those that target the critical recurrence in an application and those that do not. Using analysis and transformation, one can either entirely remove the critical recurrence, or move the recurrence so that it is no longer critical. We will see both of these cases in *LAME*.

A hand analysis shows that the first critical recurrence in the loop in lame_encoder() results from the calculations outside of the compression stage (VBR_iteration_loop()). Noise analysis (L3psycho_anal_ms()) is one of these calculations. The noise analysis sequence cannot be removed, but the code can be transformed so that the iteration-independent compression code exists in its own loop. After performing loop distribution to separate the sequential code from

[9]Functions with less than 2% of the overall runtime are not shown.

Figure 4.15 *LAME* (VBR) Callgraph

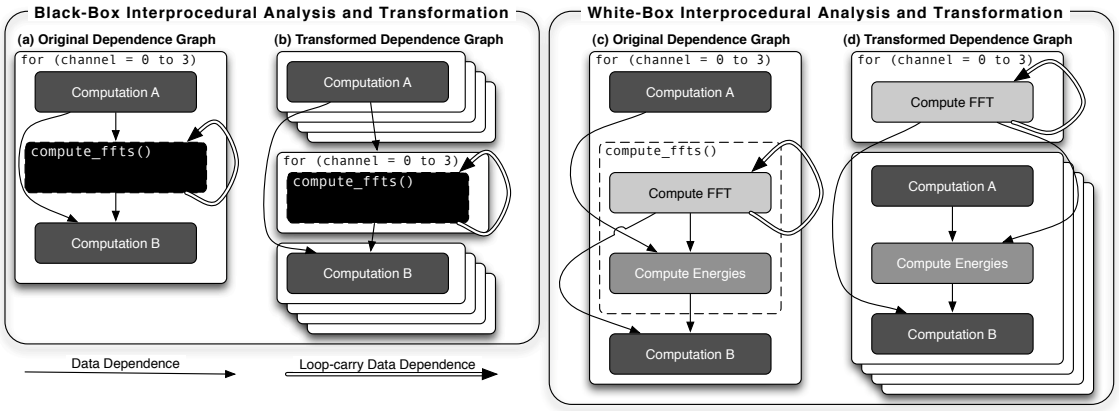


Figure 4.16 Interprocedural Loop Distribution

the parallel compression code, a different recurrence due to the bit reservoir inside of the compression sections becomes critical. Disabling this feature breaks the dependence, and moves the critical recurrence to noise analysis. As stated before, this section of the code is algorithmically sequential at the high-level. Some parallelization techniques can be used inside of different noise analysis instantiations to further improve the overall performance of the application. Both the compression and noise analysis components will be studied in the rest of this section on *LAME*.

4.2.1 Noise analysis parallelism

The noise analysis code is complicated and heavily hand optimized for performance, and therefore is difficult to analyze and transform. In L3psycho_anal_ns(), the outer noise analysis loop (psymodel.c:1337)[10] processes through the audio channels – four of them in the case of a joint stereo encoding.[11] Intuitively, the noise analysis of each channel should be independent. However, as a result of optimizations, there exist some loop-carry dependences that must be transformed out of the loop before parallelization can take place.

[10]This study uses *LAME* 3.96.1.

[11]Stereo audio is generally stored as two streams of data – one for the left channel and one for the right. It is possible to perform a loss-less translation of this type of data into a format that stores the average data for the two channels (*Mid* channel) as well as difference data (*Side* channel). In some situations, this type of data is more suitable for compression, and therefore lower bitrates can be used to encode audio with the same final sound quality. Picking between L/R and M/S for different sections of an audio stream is known as *Joint Stereo*.

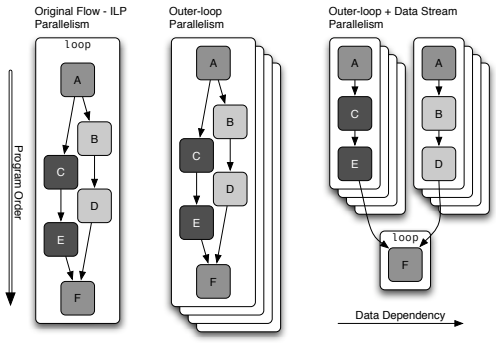


Figure 4.17 Noise Analysis Transformation after Dependence Removal

4.2.1.1 FFT interprocedural parallelism

Figure 4.16 illustrates the data dependences for an FFT calculation performed through a subroutine call. The left two diagrams show the analysis and transformation using a black-box summary of compute_ffts(), while the right two diagrams show a more detailed summary and interprocedural infrastructure.

Because the FFT data for the M/S channels can be calculated quickly by transforming the results from the R/L calculations, there exists a dependence between calls to the compute_ffts() function (the R/L loop iterations are performed first). One could change the algorithm so that the FFTs were calculated from scratch for the M/S channels – but this is outside the scope of our compiler research. The second option is to generate the FFT data before spawning the loops, and then use this data as necessary (Figure 4.16(d)). The interprocedural nature of this problem makes the necessary analysis and transformation more challenging, and this will be investigated further in Section 4.2.2.1.

There are some additional loop-carried dependences at end of the channel loop (in Computation B) that must also be removed using loop distribution in order to extract parallelism from the channel loop. These dependences are not shown in Figure 4.16, and will be ignored for simplicity in this study.

4.2.1.2 Streams of computation: Extending the parallelism

After transforming out loop-carried dependences like those in the FFT function, there is some additional coarse-grain parallelism that can be extracted from this loop. There are essentially two different strands of calculation that progress through each iteration of the loop that are independent until the end of the loop. Figure 4.17 shows a simplified example of this. The

diagram on the left shows the state of the code after the recurrences have been removed for the FFT. Two separate streams of calculation are interleaved through the code, and finally come together at the end of the processing for some final calculations. The middle diagram shows the initial implementation that takes advantage of the independent channel loop iterations. The final diagram on the right shows a different technique – one that breaks the loop into three separate components – two of which can run in parallel. There is some limited processing in block A that is shared by both streams of processing. It would have been possible to compute this a single time (similar to block F) – but because this computation is relatively simple and fast, it is better to replicate the computation than package and pass the resultant data to each thread.

While the transformations on the noise analysis fail to extract substantially scalable parallel blocks of code, they compliment the scalable transformations that can be done in compression, and therefore further improve performance. As Amdahl's law prevents the performance benefit from parallelizing the compression to about 6x, performing these transformations allows for much greater benefit out of those scalable compression transformations, decreasing the effects of Amdahl's law. Assuming that the noise analysis performance can be improved by 4x, this ultimately improves the overall performance benefit from 6x to 24x.

4.2.2 Noise analysis and transformation

As stated before, *LAME* is a very complicated application, and its analysis requirements are not near- or medium- term targets. There are, however, some important lessons that it motivates for both near- and long- term analysis and transformation developments.

4.2.2.1 FFT: Considerations for enterprocedural analysis and transformation

compute_ffts() and Figure 4.16 illustrate some transformation and analysis considerations for future interprocedural frameworks. One could conceive of building an interprocedural framework by summarizing the characteristics of different functions (Figure 4.16(a)). Using such a summary in this situation would yield analysis results that showed that it was only possible to parallelize the channel loop in two separate parallel blocks as shown in (b). One cannot hoist the entire compute_ffts() function, as this would break the dependence coming from Computation Block A. On the other hand, if the interprocedural framework enabled for more detailed block level analysis as shown in (c), it would be possible to show that the Compute FFT block could be hoisted (using loop distribution), removing the loop-carried dependence

and respecting all of the input and output dependences as shown in (d). Not only would this implementation exhibit better performance because of the reduction in thread spawn overhead and synchronization, but (b) would also require additional scalar and array expansion to handle the dependence from Computation A to Computation B.

4.2.2.2 Streams of computation: Analysis and transformation

Streams of computation (Figure 4.17) require two significant transformations. First of all, code replication is necessary for Block A. This can be performed when teasing apart the dataflow graph if Block A is small and it can be determined that precomputation and passing of the results would be as expensive as performing the code replication. Secondly, it is necessary to transform the single loop into three separate loops – one to perform each of the two streams, and one to perform the join point. Such a transformation could be performed by performing two loop distributions on the initial loop.

There are two types of analysis that one may need to perform to use streams of computation. The first type is a capability study that evaluates the dataflow and identifies the different streams as well as the join point. The second type is a study that evaluates the benefit of the transformation. While streams of computation enables additional threads to be spawned, it also decreases the amount of work that each thread performs. It will be important for the future compiler to model the different execution times of blocks of code in addition to the overhead for running them in threads. Such models will make it possible to determine the relative benefits of different transformations.

4.2.3 Compression parallelism, analysis, and transformation

Before compression, and outside of noise analysis, calculations are performed to help target the compression characteristics of different frames of audio. In VBR, these calculations incorporate the current amount of free space in the bit reservoir. For example, if noise analysis determines a certain bitrate, and there is a large amount of free space in the bit reservoir, this calculation may decide to allocate a smaller frame and use the bit reservoir to compensate. This induces a dependence between instantiations of the compression component that cannot be precomputed to allow for parallel execution. Unlike the noise analysis SCC, this SCC incorporates a large percentage of the computation performed. While it may be possible to adjust the encoding algorithm to remove the dependency and still avoid wasting space, such changes are outside the scope of our work. Instead, the compiler can create a specialized version that

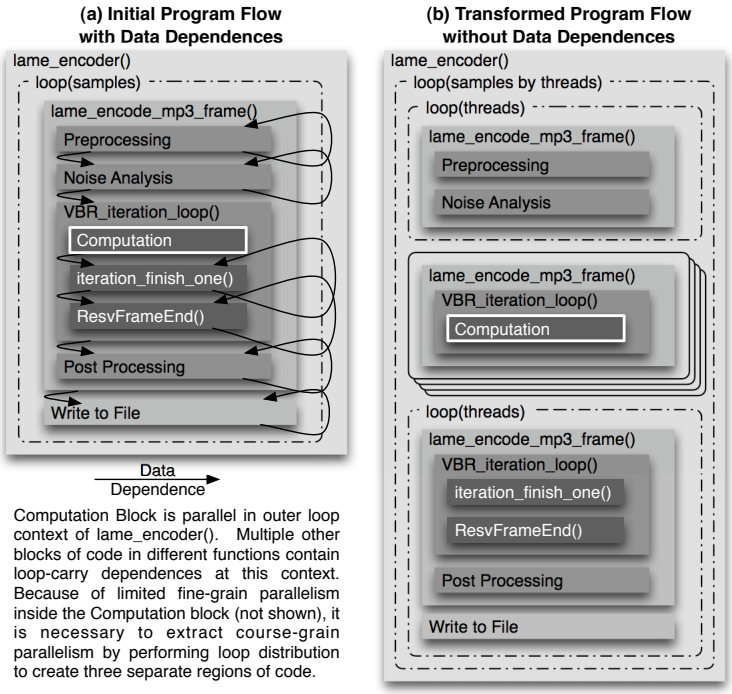


Figure 4.18 *LAME* Program Flow: Extracting Compression Parallelism

will execute when the bit reservoir is disabled. Such an implementation would allow for parallel execution of the compression blocks. It was a nontrivial task to identify this loop-carried dependence. It is possible that the analysis techniques, if they could identify this dependence with good resolution, could be used in programmer tools.

While thus far the compression block (VBR_iteration_loop) has been treated as parallel after removing the bit reservoir dependence, this is not completely true. Figure 4.18(a) presents the simplified call stack with dataflow dependence arcs drawn in. In this diagram, the Computation block, which has a white border, does not contain any loop-carried dependences around the outer loop in lame_encode() (unlike all the other blocks). A transformed version for parallel execution of the compression block is presented in (b) (data dependences are not shown to simplify the image). As we see in other benchmarks, it is necessary to perform loop distribution to hoist sequential execution prior to the parallel component into a pre-execution loop, and delay sequential execution that is after Computation to a post-execution loop. This is only possible because there are no SCCs that incorporate a large percentage of the computation. For example, if the next iteration's Noise Analysis depended on the previous iteration's Post Processing, this transformation would not be possible.

The sequential components that succeed the Computation block are at multiple call-stack depths.[12] It is unlikely that the code could be fully inlined, as there are hundreds of lines of code in many of these blocks, so IMPACT's future interprocedural framework will either have to support transformations that use specific blocks of code from different functions (like in Figure 4.16(d)), or create multiple postprocessing loops. Because the different blocks in different function calls are not dependent on each other around the back edge, it may be possible in this example to generate three separate postprocessing loops – one for each function that contains sequential code. This would require the compiler to perform additional array and scalar expansion, complicate analysis, and perform poorly in comparison to the interprocedural version.

LAME executes by passing two large structures from function to function, each of which contains pointers to large amounts of data. Determining which data to save for each thread (prior to parallel execution) and which data to retrieve from each thread (after the execution is complete) for executing the Computation block is nontrivial. In this case, it may actually be more tractable to ask the question, "Is there data that I cannot save and/or retrieve?" This is most easily motivated with an example. Let us assume that each parallel Computation block in (b) contains its own copy of these global structures and the data that they point to. After threaded execution, the successive sequential component of the loops can either copy this data into the original memory space, or access the relevant data directly. However, the loop-carried data dependence that forced these blocks to be executed sequentially is contained in data that is also stored in these large structures. As such, it is necessary to use the data generated from the sequential component and not the data saved for the threads. This situation does not manifest itself in the pre-execution loops, because the loop-carried data is never overwritten by retrieves from the thread data. In general, it is best to reduce the set of data that is saved and retrieved for the parallel components of applications. However, unlike the data that is saved, properly handling the retrieve process is necessary for correctness – simply being "conservative" and retrieving everything may result in an illegal transformation. One must be careful not to break loop-carried dependences by retrieving data from the threads. It is also important to note that determining the superset of data that must be maintained for each thread, particularly in the case of heap-allocated objects, is a nontrivial task and is left as an exercise for the reader.

[12]There are additional blocks before Computation that also are at different call-stack depths, but they are not shown in the figure for simplicity.

4.2.4 *LAME* summary

LAME does not contain substantial low-hanging fruit, but is an important benchmark for near-term development as it provides insights into some of the problems that IMPACT will face in the future. As development proceeds, these situations should be considered and planned for so that re-engineering is not necessary in the future.

LAME presents the concept of the critical recurrence. The majority of the execution time in this benchmark is spent performing the compression, but a nonnegligible time is spent performing other tasks like noise analysis. Evaluating the different loop scopes within the compression of a single frame of audio generally does not provide significant thread-level parallel opportunities, and moving out to the audio frame scope does not even suffice under many runtime conditions. After disabling the bit reservoir which removes a compression dependence, it is possible to parallelize the compression block across frames of audio. Once this is complete, the critical dependence moves to the noise analysis, which severely limits the benefits of parallelizing the compression block. Noise analysis must then be targeted to take advantage of the scalable compression parallelism.

LAME also demonstrates how programs can be used in different ways that have different runtime characteristics. Using the bit reservoir results in extremely limited coarse-grain parallel opportunities. This presents two opportunities for future research: (1) Computer Aided Design (CAD) tools for software developers to help identify potential bottlenecks in their designs and (2) compiler transformations and analysis that can optimize for a specific type of program flow.

4.2.4.1 *LAME* transformations

There are three important transformation concepts exhibited by *LAME*. First, the interprocedural framework must provide not only for analysis but also for transformations that can move specific blocks out of their original codes.[13] Second, streams of computation can be identified and extracted for additional benefit, but these generally will not produce scalable benefits. Finally, conservatively saving and restoring data for parallel execution must be performed carefully to maintain correctness.

It is not uncommon to contain function calls within loops where certain components of the function call are parallel while others are sequential. This is the case in both noise analysis with the compute_ffts() function as well as in compression in the VBR_iteration_loop(). In both

[13]The "blocks" that are refered to here do not correspond to control or basic blocks, but larger language blocks such as loops.

cases, the sequential component of the callee can be performed with the sequential components of the caller.

Streams of computation can be identified to extract additional parallelism from critical blocks of parallel code. When extracting parallelism from loops, it may be possible to identify two (or more) sets of independent work, like in noise analysis. These streams can be pulled into separate threads (sometimes with limited code replication). Despite this optimization's lack of scalability, it can be useful in cases where other techniques are not possible. It is not useful in the case of compression, as sufficiently scalable parallelism could be extracted to move the critical recurrence to noise analysis.

Finally, it is generally beneficial to identify exactly what data needs to be saved for and retrieved from parallel blocks of code. However, at times it will be difficult or impossible for analysis to provide an exact picture of what is necessary, and it is important for transformations to be conservative so that all of the input and output data of threads are transferred. One must be careful when conservatively retrieving data from threads, especially in the case of sequential postprocessing loops, as one must be careful not to retrieve data that overwrites correct data with incorrect data.

4.2.4.2 *LAME* analysis

Because of the complicated nature of *LAME*, limited time was spent exploring and presenting its analysis requirements. In general, it will be important to have a strong scalable interprocedural framework that is capable of performing field-sensitive pointer and array disambiguation analysis. Providing simple function-level summaries for interprocedural analysis about the input and output dependences will not be sufficient for extracting the best parallel implementations from applications.

4.3 *mpg123* - MP3 Decoder

The *mpg123*mp3 decoder application is generally somewhat simpler than its counterpart *LAME*, but it is still a relatively complicated application that demonstrates some interesting transforming and analysis problems. It can output directly to the audio system on a computer for the user to listen to the mp3. It can also output to a wave file, which is the mode that this study is based on. However, the transformations and analysis that we will present maintain the sequential file output, and therefore are compatible with other output formats.

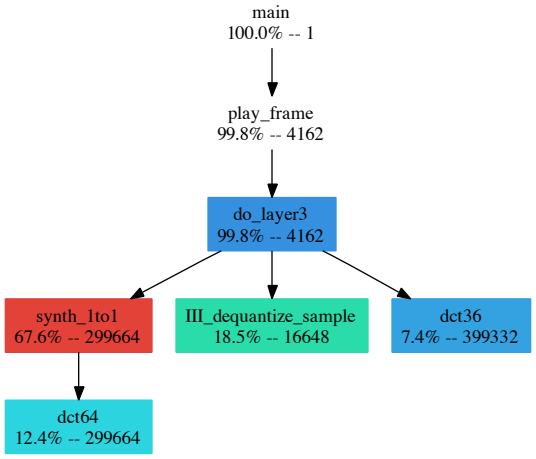


Figure 4.19 *mpg123* Callgraph with Runtime Weights

Figure 4.19 shows a pruned callgraph of *mpg123*.[14] Compared to *LAME* (Figure 4.15), this callgraph is quite simple. In this study we will focus on the synth_1to1() and dct64() function calls, which make up about two thirds of the execution time. An actual parallel implementation would have to improve the performance of the other components of do_layer3() to achieve speedups of more than 3x, but we will not investigate those other functions here.

This application works by processing through the input mp3, which is generally compressed using Huffman. After decoding the Huffman stream, a series of relatively low-runtime steps are performed, and eventually the synth_1to1() function is called. Figures 4.20 and 4.21 show the code for synth_1to1(). The first part of the function (Figure 4.20) determines the data that will be processed in the second part and calls dct64() on the input data. The second section (Figure 4.21) processes the output data from dct64(), and writes to the output buffer.

The call site of synth_1to1(), which is in do_layer3(), is wrapped in a counted loop that iterates 18 times and calls synth_1to1() 36 times (once for each of two channels in each iteration). This loop is further surrounded by another loop that iterates two times (layer3.c:1744) for every call to do_layer3. It might be possible (after some substantial transformations) to parallelize at this scope, or potentially at outermost loop in main(), but we will not investigate such implementations in this thesis.

[14] A 1 min 48 s mp3 encoded at 128 kbps is decoded by *mpg123*, a process that takes a few seconds on the x86_64 AMD Athlon 2800+ that this profile is taken on.

```
116 int synth_1to1(real *bandPtr,int channel,unsigned char *out,int *pnt)
117 {
118   static real buffs[2][2][0x110];
119   static const int step = 2;
120   static int bo = 1;
121   short *samples = (short *) (out+*pnt);
122
123   real *b0,(*buf)[0x110];
124   int clip = 0;
125   int bo1;
...
130   if(!channel) {
131     bo--;
132     bo &= 0xf;
133     buf = buffs[0];
134   }
135   else {
136     samples++;
137     buf = buffs[1];
138   }
139
140   if(bo & 0x1) {
141     b0 = buf[0];
142     bo1 = bo;
143     dct64(buf[1]+((bo+1)&0xf),buf[0]+bo,bandPtr);
144   }
145   else {
146     b0 = buf[1];
147     bo1 = bo+1;
148     dct64(buf[0]+bo,buf[1]+bo+1,bandPtr);
149   }
150
151
...   <Continued in Part 2>
221 }
```

Figure 4.20 *mpg123*:synth_1to1() Code - Part 1

4.3.1 *mpg123* parallelism

As stated above, this section will focus on the parallelism that one can extract from the synth_1to1() function call. Unlike *jpegdec*, synth_1to1() does not contain any useful inner-loop parallelism. There are a few loops in the second part (Figure 4.21) that one could parallelize, but such transformations would have limited performance benefit. While we will not go into specifics about these loops, it will be necessary to be able to summarize their access patterns for higher granularities of parallelism.

Figure 4.22 shows two possible flows for the execution of synth_1to1(). Figure 4.22(a) shows the original flow, and (b) shows a transformed version. As shown, there are no loop-carried dependences for the Computation blocks of this diagram. As such, one can execute these blocks in parallel (as shown in (b)). The outer loop calls synth_1to1() two times, once

```
 15 #define WRITE_SAMPLE(samples,sum,clip) \
 16   if( (sum) > 32767.0) { *(samples) = 0x7fff; (clip)++; } \
 17   else if( (sum) < -32768.0) { *(samples) = -0x8000; (clip)++; } \
 18   else { *(samples) = sum; }
...
116 int synth_1to1(real *bandPtr,int channel,unsigned char *out,int *pnt)
117 {
...   <Starts in Part 1>
152   {
153     register int j;
154     real *window = decwin + 16 - bo1;
155
156     for (j=16;j;j--,window+=0x10,samples+=step)
157     {
158       real sum;
159       sum  = *window++ * *b0++;
160       sum -= *window++ * *b0++;
161       sum += *window++ * *b0++;
...       <repeats>
174       sum -= *window++ * *b0++;
175
176       WRITE_SAMPLE(samples,sum,clip);
177     }
178
179     {
180       real sum;
181       sum  = window[0x0] * b0[0x0];
182       sum += window[0x2] * b0[0x2];
183       sum += window[0x4] * b0[0x4];
184       sum += window[0x6] * b0[0x6];
185       sum += window[0x8] * b0[0x8];
186       sum += window[0xA] * b0[0xA];
187       sum += window[0xC] * b0[0xC];
188       sum += window[0xE] * b0[0xE];
189       WRITE_SAMPLE(samples,sum,clip);
190       b0-=0x10,window-=0x20,samples+=step;
191     }
192     window += bo1<<1;
193
194     for (j=15;j;j--,b0-=0x20,window-=0x10,samples+=step)
195     {
196       real sum;
197       sum = -*(--window) * *b0++;
198       sum -= *(--window) * *b0++;
...       <repeats>
212       sum -= *(--window) * *b0++;
213
214       WRITE_SAMPLE(samples,sum,clip);
215     }
216   }
217
218   *pnt += 128;
219
220   return clip;
221 }
```

Figure 4.21 *mpg123*:synth_1to1() Code - Part 2

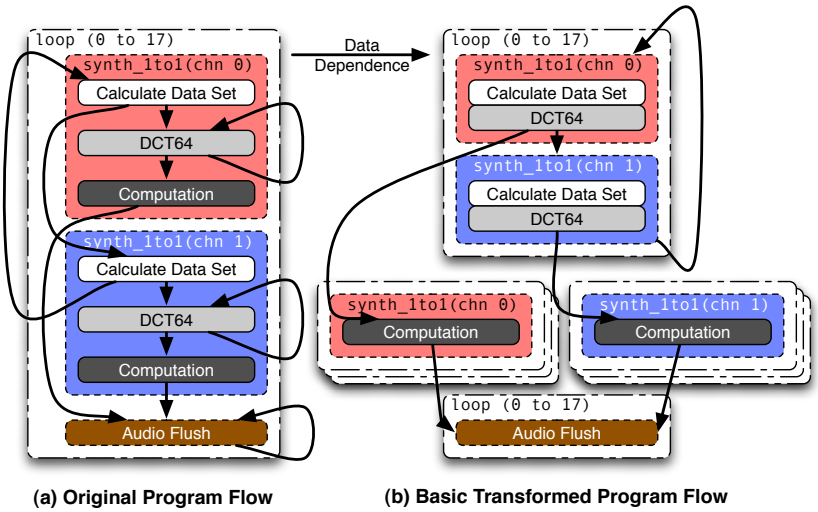


Figure 4.22 *mpg123* Program Flow - Part 1

for each channel. Because the computation for the different channels is also independent, it is possible to execute the Computation blocks for the different channels in parallel as well. The DCT64 portion updates a static array (buffs) in synth_1to1(), and the values from one iteration are used in successive executions of the Computation block. Specifically, the Computation block utilizes data generated by the most recent 16 DCT64 instantiations for each channel. Because of this, it is necessary to create a copy of the buffs array for each delayed execution of the Computation block. The buffer is relatively large (2x2x272 reals[15]), but only a portion of the buffer is necessary for each Computation block.

The Computation block contains the majority of the runtime of synth_1to1(). It is also possible to parallelize the calls to dct64(). While the benefit of performing this transformation is limited, the analysis necessary to extract the parallelism is interesting and therefore will be covered. Figure 4.23 presents such an implementation. An analysis of dct64() and the calling context will show that groups of 32 calls to the synth_1to1() function (16 for each channel) do not exhibit true memory output-dependences with respect to dct64(). Because of this, these 32 calls can be performed in parallel, while the last four for the outer loop must be done sequentially. The figure shows a slightly different implementation where each thread contains a call to synth_1to1() for each channel. Such an implementation would result in fewer threads that each performs more coarse-grained parallelism.

[15] reals are defined as either floats or doubles at compile time.

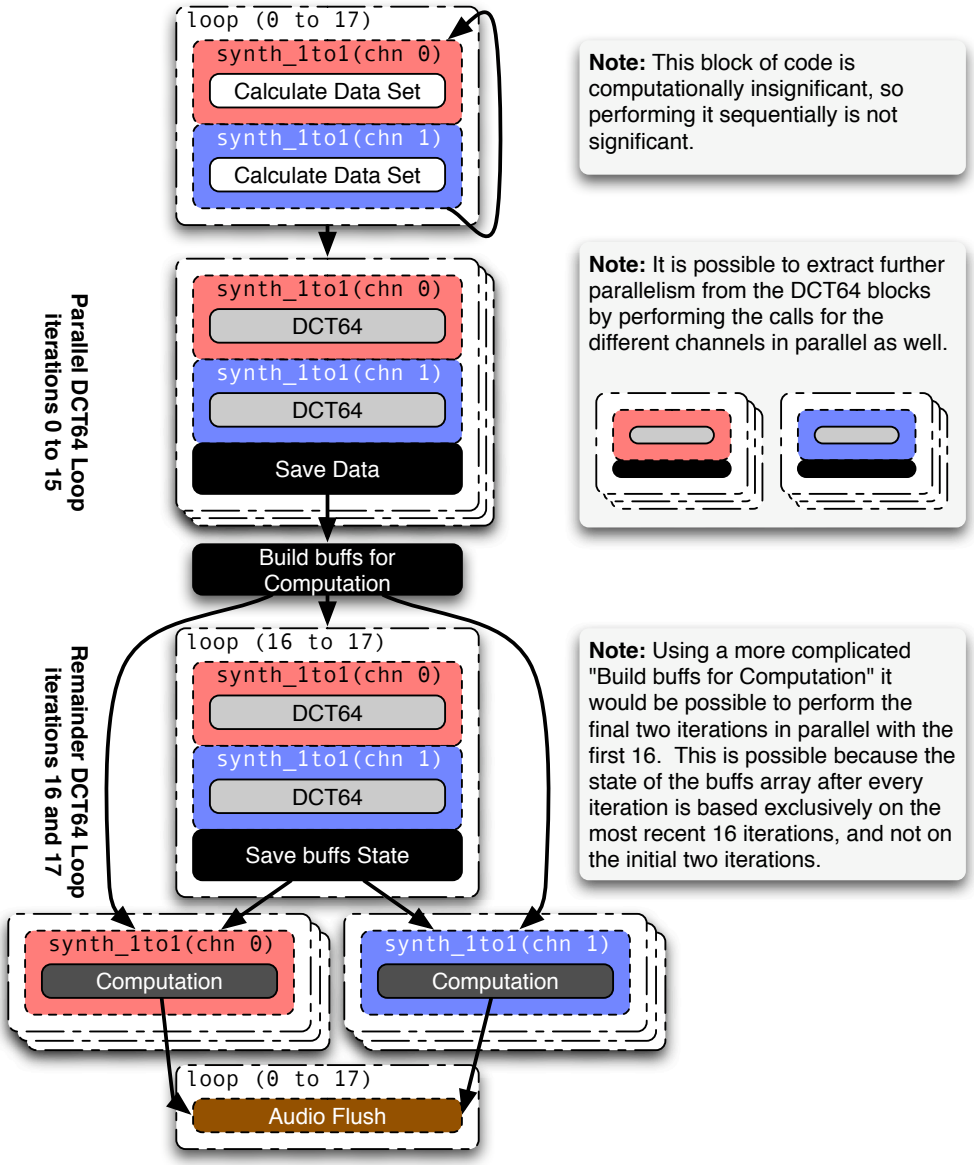


Figure 4.23 *mpg123* Program Flow - Part 2

4.3.2 *mpg123* transformation and analysis

There are two parallel implementations that we will explore in this section – the first is represented in Figure 4.22(b), and the second is shown in Figure 4.23. While the second requires substantially more challenging work for minimal performance benefit, it also provides an example for some useful analysis and transformation concepts.

The first implementation, which performs the Computation block in parallel, requires loop distribution to remove the sequential blocks of code from synth_1to1() along with the file I/O. One must also precompute the value of pnt for each call, which is based on a simple induction expression. Because the values generated by the dct64() calls in successive calls to synth_1to1() do not kill each other, and because the data used in the Computation blocks of different calls to synth_1to1() uses data generated by previous iteration's dct64() calls, it is necessary to copy the state of the buffs array for the parallel invocations of the Computation block. Only a subset of the buffs array is used for each iteration of the Computation block. This subset can be calculated by the compiler, and therefore it is only necessary to save the required subset of buffs after every dct64(). These are the only significant transformations necessary to parallelize the Computation block.

While some calls to the dct64() are actually parallel, it is only necessary to disambiguate accesses to the different Computation blocks to perform the first implementation. There are two objects that are live out of the synth_1to1() call: out, which is accessed through the samples pointer in the Computation block (line 121), and the pnt integer value, which is incremented by 128 at the end of every synth_1to1() call. It can be shown that each call to synth_1to1() will write to the space shown in Equation (4.10) (this will be left as an excersize for the reader):

$$out + (*pnt) + channel + 2 * (0\ to\ 31) \tag{4.10}$$

The out base pointer is constant and shared for the two channels. The $+channel$ and the $2*$ are used so that samples for each channel come one after each other. It can be shown that *pnt is always even. This is sufficient for proving that the data written for different channels is always distinct. 32 real values are written for each instantiation. By showing that *pnt selects a different region of the out object in successive calls, they can be disambiguated.

Figure 4.24 shows the call sites of synth_1to1() (through the fr→synth indirect function calls). The pnt value modified for channel 0 is dead, while value modified for channel 1 is used in successive iterations of the loop at line 1823. Therefore, every pair of synth_1to1() calls will have a value of *pnt that is 128 larger than the previous pair. This, combined with

```
1823     for(ss=0;ss<SSLIMIT;ss++) {
1824       if(single >= 0) {
1825         clip += (fr->synth_mono)(hybridOut[0][ss],pcm_sample,&pcm_point);
1826       }
1827       else {
1828         int p1 = pcm_point;
1829         clip += (fr->synth)(hybridOut[0][ss],0,pcm_sample,&p1);
1830         clip += (fr->synth)(hybridOut[1][ss],1,pcm_sample,&pcm_point);
1831       }
...
1842       if(pcm_point >= audiobufsize)
1843         audio_flush(outmode,ai);
1844     }
```

Figure 4.24 *mpg123*:do_layer3() Code - synth_1to1() Call Site

Equation (4.10), is almost sufficient to show that there is no dependence between calls to synth_1to1() through the data written to the output array.[16] It is also necessary for pointer analysis to be able to distinguish between the samples, window (through the decwin global pointer), and buffs objects. The existing FULCRA pointer analysis will handle these cases.

The second implementation, as shown in Figure 4.23, extends the first implementation by also parallelizing the calls to DCT64. As shown in the figure, it is not possible to fully parallelize the DCT64 calls for the outer loop, as there is an output dependence with a distance of 16 iterations for each channel. Therefore, one can perform the 16 iterations in parallel followed by the final two iterations. Because the data utilized by each Computation block is not directly produced exclusively by the most recent DCT64 call, but also include data from previous invocations, performing the array expansion is nontrivial. Each thread must write to an array, and after the parallel portion of the loop, the state of the array after each iteration must be built and saved in a sequential manner. After this is complete, the final two iterations can be executed, and their resultant states saved as well for future threaded Computation block threads.

Performing an analysis of the different dct64() calls is a challenging but tractable problem. A function-level analysis of dct64() would show two sets of output data:

[16]The object that pnt points to is global, and is modified elsewhere in the application. Specifically, the audio_flush() function at line 1843 in Figure 4.24 will reset this value to 0. This situation will not be addressed in this thesis, and is left as an open problem for the reader.

$$out0[0, 16, ...256] \tag{4.11}$$
$$out1[0, 16, ...256] \tag{4.12}$$

Every other call to this function is for a different channel. Lines 133 and 137 select the input buffer for both out0 and out1 based on the channel, and therefore each call of dct64() for the different functions is independent.

Lines 131 and 132 perform a wrap-around 15 to 0 down counter for bo for every call to synth_1to1() for channel 0 (every other call). The value of bo is used both to pick the destination location in the buffs second and third dimensions, and also to pick the b0 (not to be confused with bo). b0 is used in the second part of synth_1to1() and is not relevant for this example. There are two dct64() call sites, one where bo is even and one where it is odd. It can be shown that the odd case calls dct64() so that it modifies:

$$buffs[channel][1][(bo + 1)\&(15) + (0\ to\ 256\ by\ 16)] \quad \text{(for out0)} \tag{4.13}$$
$$buffs[channel][0][(bo) + (0\ to\ 256\ by\ 16)] \quad \text{(for out1)} \tag{4.14}$$

The even case is (almost) the exact opposite, except that it does not contain the &(15) because the wrap-around case does not exist with even numbers:

$$buffs[channel][0][(bo) + (0\ to\ 256\ by\ 16)] \quad \text{(for out0)} \tag{4.15}$$
$$buffs[channel][1][(bo + 1) + (0\ to\ 256\ by\ 16)] \quad \text{(for out1)} \tag{4.16}$$

This is convenient, as the actual set of data that is modified by the two different call sites is essentially identical.

Because bo can be shown to iterate down from 15 to 0, induction analysis with the above equations will show a dependence distance of 16. Therefore, 16 successive calls for each channel to the synth_1to1() function will result in different data being modified by the dct64() call. Because the call site of synth_1to1() is in a loop that iterates 18 times, there does exist a loop-carried dependence within this context, forcing the final two iterations for each channel to be performed sequentially.

4.3.3 *mpg123* summary

Very little useful inner-loop parallelism is available in the hot synth_1to1() function in *mpg123*. This function is called numerous times, and it is possible to perform transformations and analysis that allow for two different parallel implementations (Figures 4.22 and 4.23).

First of all, interprocedural loop distribution is necessary to extract a parallel implementation. Secondly, parallelizing the dct64() calls, despite not having a large effect on performance, is interesting because it is only possible to perform a subset of the iterations in parallel. Finally, synth_1to1() demonstrates how it is important to perform a demand-driven analysis and transformation for determining the data that should be saved for parallel invocations rather than saving entire objects.

The synth_1to1() function provides an interesting case study for a variety of reasons. First, a static buffer (which would be seen as a global object in Lcode) is used to communicate data from iteration to iteration, inserting dependences that must be dealt with in order to create a parallel implementation. Secondly, control flow provides information for the induction expression analysis, and must be considered in order to extract the dct64() calls. Finally, because it is possible to only save a subset of buffs for parallelizing the Computation block in synth_1to1(), it provides a case study for developing an analysis framework for allowing such transformations.

4.4 *MPEG-4* - Video Decoder

The *MPEG-4* video decoder application has also been studied by the IMPACT group recently. This work has been submitted for publication in [26], and will not be described in detail here. In addition to presenting details on the IMPACT compilation infrastructure, it demonstrates how context sensitivity, field sensitivity, and heap specialization must be combined together in order to disambiguate certain crucial memory accesses. It also evaluates the success of Omega Test and other array disambiguation techniques in further disambiguating critical dependences in *MPEG-4*.

4.5 *179.art* - Image Recognition

Past HPC transformations and analysis targeted benchmarks similar to those found in the SPEC floating point suites. In order to perform parallelizing compilation on pointer-rich applications like those explored previously in this chapter, it is important to build upon the wealth of past research that has gone into this class of applications.

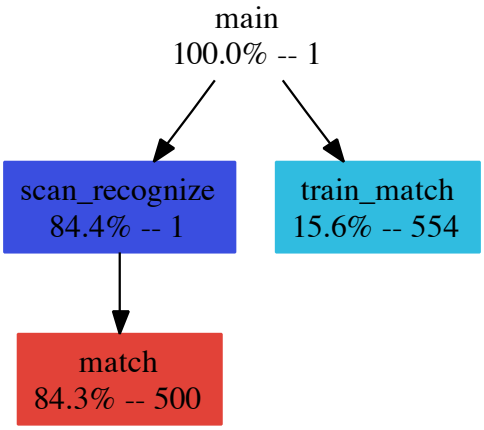


Figure 4.25 *179.art* Callgraph with Runtime Weights

179.art is an *Adaptive Resonance Theory 2* (ART 2) neural network that recognized objects in a thermal image based on training data. Figure 4.25 shows the high-level pruned callgraph of the application with execution times.[17] As the figure shows, the majority of the runtime is spent in the match() function.

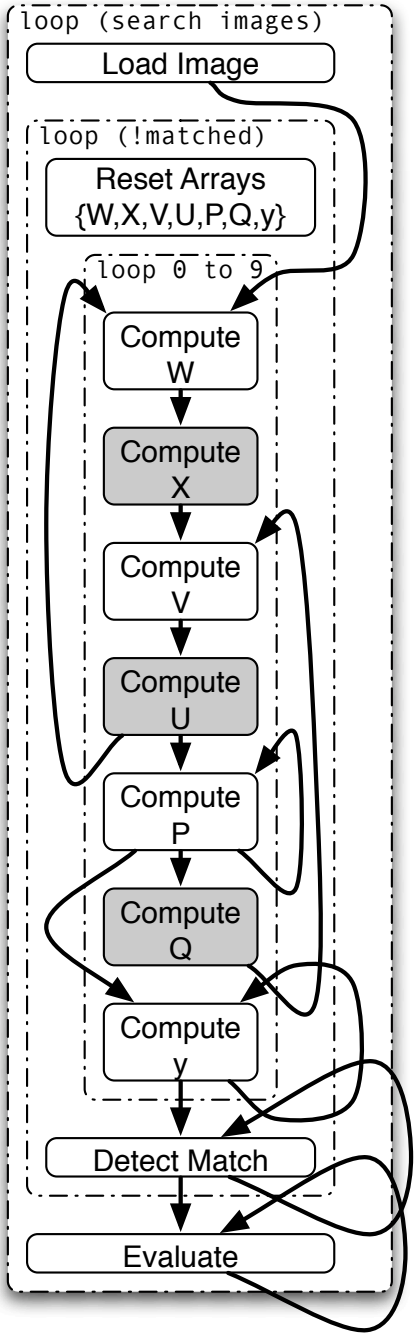
4.5.1 match() overview and parallelism

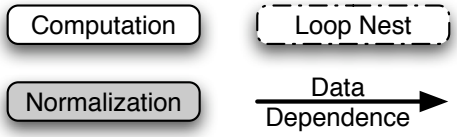
179.art executes by first performing training with match_train(), and then attempts to find an object in a search image by iterating through regions of it and calling the match(). The majority of the runtime is spent performing the actual search, and in general the characteristics of match_train() and match() are very similar. Therefore, we will focus on the match() function and its call site in this section.

Figure 4.26 provides an algorithmic view of the match() function and its calling context in main(). Central to the processing within match() are a series of blocks of computation, each of which is either 1 or 2 loop nests. Almost all of these blocks are dependent upon the previous block (in program execution order). Outside of these blocks are a series of additional loops in the calling context.

The inner loops are all high-iteration count loops with a large amount of parallelism. The computation loops each contain an accumulator that is used in the successive normalization loops, but these can easily be removed with accumulator expansion. Because of the normalizations, it is not possible to perform large-scale loop fusion despite the fact that the loops share

[17] This was collected on an x86_64 AMD 2800+ system compiled with gcc (3.4.2) using -pg -O2. This shows the profile for input3.





Fine-Grain Parallelism
Each computation or normalization block represents 1-2 loop nests with high iteration counts that contain no loop carry dependences (in the case of the normalization blocks) or accumulators that can be fused out (in the case of the computation blocks) in order to achieve fine-grain parallelism.

Limitted Loop Fission
Because of the accumulators that are calculated for normalization, it is not possible to fuse all of the inner loops. However, the normalization loops can be fused with the successive computation loops.

Medium-Grain Parallelism
The loop-back dependences that are drawn show the loops that the dependence carries between. As shown, the `0 to 9` loop is heavily dependent upon past iterations, and therefore cannot be transformed for parallel execution. The `matched` loop, on the other hand, is only contains a dependency for the final computation block, and therefore can be made parallel. This is not a counted loop, and tends to iterate a limited number of times, and therefore is a poor opportunity for parallelization.

Course-Grain Outer-Loop Parallelism
There are some sequentializing dependences in for the outer loop. Transforming these out and performing privatization of the inner loop arrays, one can perform the outer loop in parallel.

Reset Arrays
Before performing computations, the various global arrays are reset to zero, effectively killing all past iteration computation. This is useful, as it removes many dependences for the outer two loops.

Figure 4.26 *179.art*:match()- High-Level Flow

the same loop bounds. However, it is possible to fuse each normalization loop with the computation loop that follows it. Performing this transformation slightly increases the granularity of the parallelism, and may improve performance.

The first loop outside of the inner computation loops, shown as *loop 0 to 9*, is a convergence loop that is limited to 10 iterations (there is an additional loop-bound condition that is not shown). It can break out prior to the 10th iteration if the resulting P array is the same as the previous iteration. This loop contains numerous loop-carried data dependences, and is not a strong candidate for parallel execution.

The !matched loop iterates until it is confident that it has converged to the likelihood that the current search space contains the search object. This loop contains only a single loop-carried dependence, and therefore could be transformed to execute in parallel. However, the number of loop iterations is dependent upon the calculations that are performed in each iteration of the loop. Because of this characteristic, this loop is not a strong candidate for parallelization.

The outer loop that calls the match() function (which is actually two nested loops) processes through the entire search space and selects subregions to search for the image within. Each of the calls to match() is independent (after some minor transformation that we will explore in Section 4.5.2.2), and because of the high iteration count of these loops, they present an excellent opportunity for scalable coarse-grain parallelism.

179.art is a strong test benchmark for future developments. It contains simple fine-grain parallelism that will provide a useful method for future developments, as well as simple coarse-grain parallelism that is far more scalable than simple inner loop parallelizing techniques.

4.5.2 match() analysis and transformations

4.5.2.1 Inner loop fine-grain analysis and transformation

There are generally three types of parallelizable inner loop nests. First, there are normalization loops (Figure 4.27), which consist of a single loop nest and simple code. Next, there are computation loops (Figure 4.28) that consist of a single loop. Finally, there are doubly-nested computation loops (Figure 4.29).

The normalization loops (X, U, and Q from Figure 4.26) do not require any substantial transformation or analysis. Figure 4.27 shows one of the normalization loops. f1_layer[tj].X and f1_layer[tj].W can be disambiguated using struct offsets in the existing pointer analysis. Once this is complete, it is trivial to prove that each iteration is independent.

```
551       /* Compute F1 layer - X values */
552
553       for (tj=0;tj<numf1s;tj++)
554             f1_layer[tj].X = f1_layer[tj].W/tnorm;
```

Figure 4.27 *179.art* - Normalization Loop

```
543       /* Compute F1 layer - W values */
544       tnorm = 0;
545       for (ti=0;ti<numf1s;ti++)
546       {
547           f1_layer[ti].W = f1_layer[ti].I[cp] + a*(f1_layer[ti].U);
548           tnorm += f1_layer[ti].W * f1_layer[ti].W;
549       }
550       tnorm =  sqrt((double)tnorm);
```

Figure 4.28 *179.art* - Single-Loop Computation Loop

```
580       /* Compute F1 layer - P values */    ***** P COMPUTATION *****
581       tnorm =0;
582       tsum=0;
583       tresult = 1;
584       for (ti=0;ti<numf1s;ti++)
585       {
586         tsum = 0;
587         ttemp = f1_layer[ti].P;
588
589         for (tj=0;tj<numf2s;tj++)
590         {
591            if ((tj == winner)&&(Y[tj].y > 0))
592                  tsum += tds[ti][tj] * d;
593         }
594
595         f1_layer[ti].P = f1_layer[ti].U + tsum;
596
597         tnorm += f1_layer[ti].P * f1_layer[ti].P;
598
599         if (ttemp != f1_layer[ti].P)
600            tresult=0;
601       }
602       f1res = tresult;
...
610       /* Compute F2 - y values */           ****** Y COMPUTATION *****
611       for (tj=0;tj<numf2s;tj++)
612       {
613         Y[tj].y = 0;
614         if ( !Y[tj].reset )
615         for (ti=0;ti<numf1s;ti++)
616            Y[tj].y += f1_layer[ti].P * bus[ti][tj];
617       }
```

Figure 4.29 *179.art* - Doubly-Nested Computation Loops

The singly nested iteration computation loops (W and V from Figure 4.26) are similar to the normalization loops, in that only existing pointer analysis and basic array disambiguation is necessary, except that these loops accumulate a normalization constant for their successive normalization loops. Figure 4.28 shows an example of this. Basic accumulator expansion can be used to remove these dependences from the loop.

The doubly nested computation loops (P and y from Figure 4.26) are an extension to the singly nested loops and are not much more complicated. In each case, the outer loops iterate the same number of times as both the singly nested computation loops and the normalization loops. In P (Figure 4.29), the inner loop calculates tsum based on the tds global array. Once the inner loop has finished iterating, P is calculated based on tsum and the previously computed U array. The inner loop is an accumulation loop, and therefore could be parallelized using accumulator expansion. It is not necessary to extract parallelism from this inner loop, but the loop does provide a useful test case for accumulator expansion. After performing accumulator expansion on the tnorm calculation in the outer loop, the outer loop can be shown to be completely parallel after the f1_layer and tds global arrays are shown to be disjoint.

The y computation loop (Figure 4.29) is slightly different from the P computation. In it, the y is actually accumulated in the inner sequential loop from f1_layer and bus (all global arrays). After these arrays have been disambiguated, it is trivial to show that all outer loop iterations are independent. No transformations like accumulator expansion are necessary.

Simple loop distribution is possible between the normalization loops and their successive computation loops. With SSA that supports globals (numfls1), it is possible to show that all of the loops have the same bounds. Once this is complete, induction expression analysis can show that the array element generated by the i^{th} iteration of the normalization is used in the i^{th} iteration of the following computation loop. After performing distribution, it is possible to remove the store instructions for the normalization arrays, as the results of the computation are dead after their use in the computation loops. This would be a complicated optimization, as it would be difficult to prove because the arrays are global. Register promotion would decrease the critical path of the computation. One problem with register promotion in a multithreaded environment is that it becomes more difficult to prove that no other memory operations will modify a value between a load and a store (because another thread could modify it). Because our compiler is performing the parallelization, it is possible to make assumptions about the memory characteristics of different threads. In general, parallel threads that are created by the IMPACT framework (and parallel programs in general) should never modify shared data

```
1010     for (j=starty;j<endy;j=j+stride )
1011        for (i=startx;i<endx;i=i+stride)
1012        {
1013          k=0;
1014          for (m=j;m<(lheight+j);m++)
1015            for (n=i;n<(lwidth+i);n++)
1016                f1_layer[k++].I[0] = cimage[m][n];
1017          pass_flag =0;
1018          match();
1019          if (pass_flag==1)
1020          {
...
1024             if (set_high[0]==TRUE)
1025             {
1026               highx[0] = i;
1027               highy[0] = j;
1028               set_high[0] = FALSE;
1029             }
1030             if (set_high[1]==TRUE)
1031             {
1032               highx[1] = i;
1033               highy[1] = j;
1034               set_high[1] = FALSE;
1035             }
1036          }
...
1041        }
```

Figure 4.30 *179.art* - match() Call Site

outside of critical regions, with the exception of lock acquisition and release. Unless speculation capabilities that support detection and roll-back are implemented, future transformations should never be made unless this characteristic can be proved and maintained.

4.5.2.2 Outer loop course-grain analysis and transformation

Figure 4.30 shows the code that calls match(). As an aside, this code is broken. The code assumes that set_high, highx, and highy are only two elements (as they are declared). However, the code in match() modifies set_high as if its size is numf2s - 1 - 1, where numf2s is calculated by objects + 1, where objects is based on the user input. If the user input was verified, this code would be valid.

The match() call site works by calling the match() function for different regions of the search image and calculates a confidence of a match for each section. If the confidence of the current section is higher than past confidence values, then the code marks to save that location in the image by setting the set_high flag, which is later processed by the calling function. After performing one of the match() calls, the set_high flags are checked and then reset. The problem

here is in the reset. Rather than performing the reset inside of the match() call, or potentially above the match() call, a conditional loop-carried dependence is created. This problem can be solved by removing the conditional for the flag sets and pushing them around the back-edge above the match() call.

The next loop-carried dependence that must be handled also involves the set_high flags. The code that conditionally sets these flags (lines 644 to 648) must be sequentialized. This code is inside a series of if nests inside match() that are also used to determine if the loop at line 536 is complete. By saving the match_confidence value for successive traces through this code, one can move this code outside of the line 536 loop. It can be placed above the code that checks the flags at line 1024 (after the match() call site). This transformation accumulates all of the sequential code to the end of the loop 1010/1012 nests, and can then be removed from those loops using loop distribution.

There are two types of analysis that are necessary for performing the parallelization described above – those that disambiguate the truly independent sections of match(), and those that identify the loop-carried dependences that must be sequentialized. There is a true loop-carried dependence for the highx and highy values at the end of the call site loops. These are easy to identify. The next loop-carried dependence exists with the highest_confidence array at line 646. Because of the way the application has been coded, there also exists a dependence through set_high for lines 647, 1024, 1028, 1030, and 1034. In general, it is not a problem for compilers to identify dependences because of the conservative nature of their analysis frameworks. As such, these dependences already exist in IMPACT.

Disambiguating the parallel components of match() is slightly more difficult (as one would expect). *179.art* uses global arrays and scalars to maintain the state of the system – and this may make analysis challenging, as many analysis techniques do not work well with global values. At the beginning of the match() call, all of the array values are reset with a call to reset_nodes(). This call loops through numf1s elements of each of the arrays that are generated during the processing. Each processing block iterates the same number of times, so an induction analysis that incorporates SSA on global variables will show that the values that are generated by each call to match() are killed at the beginning of successive calls, and therefore no true dependence exists between iterations. The identified output-dependence between previous iterations and the call to reset_nodes() must be removed by using array expansion on the global arrays. Because the array is global, it is necessary to guarantee that the correct array is live out of match(), as it could be access later in the application (this is not the case in *179.art*).

4.5.3 *179.art* summary

179.art is generally a great application for near-term benchmarking of both transformation and analysis developments. It contains opportunities for both simple fine-grain parallel execution and relatively simple coarse-grain extraction. However, while working with IMPACT to develop transformations and analysis, one must be careful not to develop solutions that are restricted to the simple cases presented in *179.art*. Unlike many of the other benchmarks, *179.art* uses global data for much of its communication between functions. As such, it is a good benchmark for testing IMPACT's ability to deal with global structures and pointers.

4.5.3.1 *179.art* transformations

Simple inner-loop parallel execution requires accumulator expansion and can benefit from simple loop distribution. These transformations should be implemented in the near future. Moving out to the coarse-grain parallelism will require loop distribution. These transformations must be performed across function boundaries, so it will ultimately be a good benchmark of interprocedural transformations. Unlike in other benchmarks like *LAME*, it will be possible to work with an inlined version when performing transformations (and analysis). This will be useful as it will allow both for near term development in coarse-grain transformations (without the interprocedural framework) and for a comparison between the results of the interprocedural framework and those of an inlined version.

4.5.3.2 *179.art* analysis

Many of the analyses that are necessary for identifying opportunities for parallel execution in *179.art* have already been implemented in IMPACT. Because *179.art* largely works with global structures, it will be a good test of existing analyses, including SSA and induction expression analysis, to determine if they work properly with globals. This work is currently underway. In addition to this, the coarse-grain parallel extraction will require gen-kill on arrays that take into account loop bounds on global variables that are constant at run-time.

CHAPTER 5
IMPACT ROADMAP

This chapter presents a roadmap for future IMPACT development. It is broken into four sections. Section 5.1 presents some "low-hanging fruit" for both analysis and transformation. These are techniques that should be worked on in the near term. Section 5.2 provides an overview of some of the requirements and considerations for the interprocedural analysis and transformation framework that will be developed. Finally, Sections 5.3 and 5.4 identify tasks for medium-term transformations and analysis (respectively).

5.1 Low-Hanging Fruit: Analysis and Transformations

This section will identify some near-term tasks for both analysis and transformations that should be developed in parallel with the interprocedural framework (Section 5.2). It will be important to complete many or all of these tasks prior to moving on to the more complicated ones presented in the Sections 5.3 and 5.4. This section will begin with some basic transformations that are driven by pragmas. These subsections will also contain comments on developing and evaluating the analysis framework with respect to the transformations. Next, it will motivate some enhancements that can be made to FULCRA. It will conclude with a discussion on how the analysis framework can be evaluated beyond what was discussed with the transformations.

5.1.1 Parallel IMPACT using pragmas

Automatic identification of locations to perform parallel execution is very difficult, and is not a task that will be completed in the near term. As such, this section will assume that the code (whether C or Lcode) has pragmas to identify locations where parallelism should be extracted. This instrumentation will essentially be an extension similar to OpenMP. The user will identify code to parallelize and the compiler will analyze this code and transform it for legal parallel execution. This is different from OpenMP, which assumes that identified loops do not contain loop-carried dependences. We will focus on *jpegdec*, *mpg123*, and *179.art*. *MPEG-4* will likely also benefit from these transformations. *LAME* is more complicated than these other benchmarks, and in general will be ignored in this section.

5.1.1.1 Basic thread extraction

There is currently work underway to perform loop blocking to convert single loops into multiple loops. Recall that this transformation can convert a loop that, for example, iterates 60 times and transform it into a doubly-nested loop that iterates 6 times (outer loop) and 10 times (inner loop). The outer loop can then be used to spawn 10 independent threads, each of which executes the inner loop. This transformation is one of the most basic transformations, and will be used frequently (commonly after other transformations have removed any pre-existing SCCs from the dataflow graph). Loops without loop-carried dependences exist in both ycc_rgb_convert() in *jpegdec* and match() in *179.art*, and these should be used to test this transformation. This transformation should also be tested with the Lpar transformation that spawns threads off of a single loop nest.

5.1.1.2 Accumulator expansion and loop fusion

The match() function in *179.art* contains multiple loops with simple accumulators. An analysis to identify these cases and perform accumulator expansion on them should be implemented. Once this is complete, these same loops (which become parallel) can have loop fusion performed on them as described in Section 4.5.2.1. There are two steps to implementing the loop fusion transformation. The first step should utilize pragmas to identify two loops that can be combined. From this, a transformation can be built alongside an analysis that verifies that it is legal. This can then be extended so that the programmer marks a function to search for loop fusion opportunities. Loop nests can then be pair-wise compared and evaluated (using the analysis from the first step) for loop fusion.

5.1.1.3 Removal of scalar loop-carry dependences

The h2v2_fancy_upsample() function in *jpegdec* contains scalar loop-carried dependences in its innermost loop (Section 4.1.4.1). In order to perform the transformation described in Section 5.1.1.1, it is necessary to remove this dependence. Because the dependence is based on a calculation that can be pulled around the edge of the loop, code expansion can be applied to remove the loop-carried dependence from some iterations.

```
1: procedure LOOP ALLOCATION EXAMPLE
2:     for all i in elements do
3:         Object[i] ← malloc(sizeof(Object))
4:     end for
5: end procedure
```

Figure 5.1 Loop Allocation

5.1.2 Memory allocation

It is generally good practice to use *allocation pools* to replace malloc in applications, as it tends to improve performance. Allocation pools malloc large chunks of memory and provide pieces of it to the application as is necessary. As a result, all objects that are allocated with a specific allocation pool will all be assigned to the same initial points-to set. Heap cloning will not be able to disambiguate these allocations even though the allocation pools may have different call chains because malloc is not invoked separately for each call. It is possible to mark functions other than malloc as allocation routines when providing the input to FULCRA. By using pragmas to mark pool allocation functions as "malloc-like," and adding support to FULCRA to use these pragmas, additional spurious dependences will be removed. It may later be possible to build an analysis in Pcode that identifies these pools, but in the short-term user annotations should be an acceptable solution.

In C, multidimensional arrays are compiled as single dimensional arrays that cover a large block of memory. These semantics guarantee that the data with different dimensions never overlaps. As described in Section 4.1.6, multidimensional arrays in *jpegdec*are allocated as arrays of pointers to arrays instead of as a single block of memory.[1] Functionality should be added to mark the functions that allocate these multidimensional pointer arrays. Later it may be possible to create an analysis that can identify them, but this should be sufficient for the short-term.

Support also needs to be added that can disambiguate objects that are allocated inside of a loop as shown in Figure 5.1. In this case, we have a single malloc call site that allocates a series of distinct objects. This is very similar to the case above. At the time of allocation, it is easy to see that each of the objects pointed to in the "Object" array are independent. If it can be proved that these pointers are never modified, then the objects pointed to by different indexes must be independent.

[1] Although it is not covered in this thesis, this situation manifests itself in *MPEG-4* as well.

5.1.3 Analysis evaluation

In addition to creating analysis to support the transformations identified above, it is important to evaluate the performance of existing pointer and array disambiguation analysis before moving on to coarse-grain parallelism. The iDCT function in *jpegdec* and dct64() in *mpg123* should be evaluated because they modify noncontiguous parts of their output buffers (i.e., every 16th entry). match() in *179.art* should be evaluated for the summary information across the different loop bodies.

5.2 Interprocedural Framework

As people begin to develop the necessary infrastructure described in Section 5.1, others should begin to develop the interprocedural framework. The majority of the topics that we will be addressing in the previous section are not new or novel ideas. Although it may be possible to publish them in the context of the more realistic benchmark suite, having a scalable and effective interprocedural analysis and transformation system will be an important contribution. Most past parallelizing work has dealt with simple kernels of computation and toy benchmarks. Extending these ideas to function under the constraints of large and complicated software codebases is an important contribution to the field.

The "interprocedural framework" will likely consist of two separate and distinct frameworks – one for analysis and one for transformation. This section provides benchmark-driven insights into some of the requirements of both of these infrastructures as design commences.

5.2.1 Interprocedural analysis framework

It will not be sufficient to provide function-level summaries to extract high-performance. There are a variety of options for performing a more accurate analysis. The first option, which is less scalable but the most accurate, would be to emulate inlining the function(s) and then perform the analysis. This would provide accurate results, but likely would run into the same scalability problems that full program inlining hits. The second option is to provide region-based summary information for each function. While it may be possible to provide summary information of all of the loop-nests and sequential blocks of code, it may be difficult to work with. The compute_ffts() function from *LAME* requires summary information of the outermost loop nests. This is the case for match() in *179.art* as well. If possible, it may be interesting to develop a framework that can provide varying levels of summary information. It may also

be useful to provide detailed summary information, but tailor the analysis to use subsets of the summary information to enhance scalability.

In *LAME* parallelizing the compression requires the identification of the parallel "Computation Block" (Figure 4.18) many function calls down from the actual loop through which it is parallel. This demonstrates how it will not be sufficient to limit the number of function calls that the interprocedural framework can process through.

Another consideration for any function summary information that is generated will be how to work with the function parameters. FULCRA may have decided that two of the parameters access the same object. There may be cases where array disambiguation that utilizes caller loops is able to determine that the uses of the two parameters will never overlap in the function. If the summary information merges these parameters in the summary before array disambiguation is applied from the calling context, then it may never be possible to disambiguate the different objects. At the same time, FULCRA may have determined that the parameters to a function alias because of operations within that function and not because of the characteristics of the callee. If this is the case, then losing this information in the summaries, and potentially allowing array disambiguation to determine that the two parameters do not alias, would also be incorrect. Developing a system that handles these two cases is an interesting research topic.

5.2.2 Interprocedural transformation framework

Transformation opportunities exist that call for the movement of specific blocks of code across procedural boundaries. As described in an example in Section 4.2.1.1, it is sometimes necessary to move specific sections of a function into another function, leaving others intact. Not only will it be necessary for the interprocedural framework to maintain the information necessary for such transformations, but it will also need to support transformations that move specific blocks of code from different functions.

5.3 Transformation

While Section 5.1 provided some transformations for short-term development, this section will summarize some of the medium- and long-term transformations that will be necessary to extract more coarse-grained parallelism from some of the applications. We will start with some of the medium-term targets, and finish with more complicated long-term tasks.

One task will be to extend the traditional transformation techniques to support nested loops, as well as those with more complicated control flow. In *jpegdec*, iDCT is called from nested

loops (3 deep) that iterate a total of six times. Parallelizing at any of these levels alone will result in limited performance benefit. In *179.art*, match() is called from a set of doubly nested loops that iterate through the dimensions of the search image. In this case, the coarse-grained parallelism that can be extracted from either loop alone is sufficiently scalable. However, parallelizing the outer loops means that the sequential component, which is contained inside both of the loops, must be pushed down below both loop nests rather than just the one (see Section 4.5.2.2 for more details). In the h2v2_fancy_upsample() function in *jpegdec*, extracting medium-grain parallelism that crosses the call-site requires extracting threads from nested loops where the outer loop exists in the caller and the inner in the callee. In the front-end of *jpegdec*, when extracting coarse-grained parallelism, loop distribution must be performed on a set of nested loops that contain a return statement that breaks out of the loops (Section 4.1.3.4).

Many of the medium-grain transformations require loop distribution across function boundaries. There are cases of this in the front- and back-end of *jpegdec*, in match() in *179.art*, and in synth_1to1() in *mpg123*. In many of these cases, it is feasible to inline the functions and develop the transformations to handle them before the interprocedural framework is complete. As the framework begins to come online, it will then be possible to adapt the existing transformations to work with the framework and necessary to evaluate the results with and without inlining.

The existing inlining infrastructure can inline through indirect function calls. Because of the heavy use of indirect function calls in these applications, it will be necessary for the transformations to function in a manner similar to what is done in Pinline. The characteristics of an indirect function call must be conservative for all possible called functions. Converting an indirect call into control flow with a direct call (like in Pinline) may also allow for less restrictive analysis results. By using code expansion, this control flow can be pushed above loops (assuming it is not dependent on the loop iteration) and the entire loop nests can be specialized. In the back-end of *jpegdec*, the indirect upsampling calls are dependent on the loop iteration (different components use different upsampling routines). In this case, unrolling this loop would allow for specialization of the different component's function calls, which may help to enable the analysis and transformations.

Much of the analysis performed in this thesis makes certain assumptions about the mode in which the different applications are executed. As transformations are made, it will be important to take advantage of the special characteristics of different modes, and optimize accordingly. At the same time, it will be necessary to ensure that all other modes will still function properly. For example, sep_upsample() cannot be optimized for all program inputs assuming that

h2v2_fancy_upsample() will be called for both the Cr and Cb frames. However, it may be useful to create a specialized version of sep_upsample() for when this assumption is true.

One can extend these path specialization techniques to handle situations where certain characteristics cannot be proved by the analysis (or even the programmer). For example, if certain transformations require that loop bounds be constant, but it is not possible to prove that they are, one could instrument the code with checks to verify this assumption and transform accordingly. Such functionality would also be useful in cases where characteristics were generally true but occasionally false. If the benefits of achieving parallelism outweigh the penalty for handling the false case, then these transformations can and should be made. This is similar to explicit data speculation in EPIC compilation.

In general, many of the transformations that are performed reorder the execution of code by transforming the control flow. At times, it may be difficult to perform these transformations. If an analysis can prove that there are no true loop-carried dependences between different blocks of code in a loop, one can reorder execution by creating continuations not only of data but also of functions. The coarse-grain model of the back-end of *jpegdec* (see Section 4.1.4.3) does a good job of illustrating this concept. While this type of transformation may simplify transformation and be more general, it requires the same difficult analysis.

5.4 Analysis

Much of the necessary analysis work to be done involves extending the existing array disambiguation techniques to work interprocedurally and working through corner cases. There are some additional new analysis techniques that will be necessary to allow the existing techniques to work in general purpose codes.

179.art shows that it will be necessary to handle variable loop bounds that are constant within a certain contexts. It will be necessary to do this interprocedurally as well. In *179.art*, when match() is parallelized through the loop at its call site, it is necessary to determine that the loop bounds of each of the computation loops are constant across these loop iterations. Since SSA is capable of handling this intraprocedurally, it will need to be extended to the interprocedural infrastructure.

In both *jpegdec* and *MPEG-4*, performing value-flow analysis to determine the relationship between different variables will be necessary to disambiguate different array accesses. While it may not be possible to determine the actual values of certain variables, it is sometimes possible (and necessary) to determine relationships between sets of variables. Feeding these

relationships into the array disambiguation analysis will allow for it to remove blocking spurious dependences. There has been some past work in interprocedural value-flow [27, 28], but this was not done in the thread-level parallelism context.

REFERENCES

[1] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, "Field testing IMPACT EPIC research results in Itanium 2," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 26–39.

[2] International Technology Roadmap for Semiconductors, 2005, http://public.itrs.net.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann Publishers, 1996.

[4] Cray Research, Inc., *CF77 Compiling System, Volume 1: Fortran Reference Manual*, 1990.

[5] R. Gupta, S. Pande, K. Psarris, and V. Sarkar, "Compilation techniques for parallel systems," *Parallel Computing*, vol. 25, nos. 13,14, pp. 1741–1783, 1999.

[6] Standard Performance Evaluation Corporation, "SPEC CFP2000 benchmarks," 1999, http://www.spec.org/cpu2000/CFP2000.

[7] U. Banerjee, *Loop Transfomations for Restructuring Compilers*. Boston, MA: Kluwer Academic Publishers, 1993.

[8] S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann Publishers, 1997.

[9] W. Pugh, "The Omega Test: A fast and practical integer programming algorithm for dependence analysis," in *Proceedings of Supercomputing 1991*, November 1991, pp. 4–13.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[11] W. Baxter and I. H. R. Bauer, "The program dependence graph and vectorization," in *Conference Record of the 16th ACM Symposium on the Principles of Programming Languages*, January 1989, pp. 1–10.

[12] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard, "Value dependence graphs: Representation without taxation," in *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL)*, 1994, pp. 297–310.

[13] P. Tu and D. Padua, "Gated SSA-based demand-driven symbolic analysis for parallelizing compilers," in *Interational Conference on Supercomputing*, 1995, pp. 414–423.

[14] X. Kong, D. Klappholz, and K. Psarris, "The I-Test: An improved dependence test for automatic parallelization and vectorization," *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Parallel Languages and Compilers*, vol. 2, pp. 342–349, July 1991.

[15] R. Wilhelm, S. Sagiv, and T. W. Reps, "Shape analysis," in *Proceedings of the 9th International Conference on Compiler Construction*, March 2000, pp. 1–17.

[16] R. Ghiya and L. J. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in c," in *Symposium on Principles of Programming Languages*, 1996, pp. 1–15.

[17] J. W. Sias, "A systematic approach to delivering instruction-level parallelism in EPIC systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[18] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[19] E. M. Nystrom, "Fulcra pointer analysis framework," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[20] J. Player, "An evaluation of low-overhead partial flow-sensitivity," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2005.

[21] J. D. Choi, M. G. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, January 1993, pp. 232–245.

[22] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 54–61.

[23] S. Zee Ueng, "Template bundling for EPIC architectures," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2004.

[24] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "Memory system performance of programs with intensive heap allocation," to appear in *ACM Transactions on Computer Systems*, 2006.

[25] P. Feautrier, "Array expansion," in *ICS '88: Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 429–441.

[26] S. Ryoo, R. E. Kidd, C. I. Rodrigues, S.-Z. Ueng, M. I. Frank, and W. W. Hwu, "Interprocedural analysis for coarse-grain parallelization of C media programs with heap usage," submitted to *Proceedings of the ACM SIGPLAN '06 Conference on Programming Language Design and Implementation*, 2006.

[27] R. Bodik and S. Anik, "Path-sensitive value-flow analysis," in *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, January 1998, pp. 237–251.

[28] R. Bodik, "Path-sensitive, value-flow optimizations of programs," Ph.D. dissertation, University of Pittsburgh, Pittsburgh, PA, 1999.