Abstract

SO, WON. Software Thread Integration for Converting TLP to ILP on VLIW/EPIC Architectures. (Under the direction of Alexander G. Dean.)

Multimedia applications are pervasive in modern systems. They generally require a significantly higher level of performance than previous workloads of embedded systems. They have driven digital signal processor makers to adopt high-performance architectures like VLIW (Very-Long Instruction Word) or EPIC (Explicitly Parallel Instruction Computing). Despite many efforts to exploit instruction level parallelism (ILP) in the application, typical utilization levels for compiler-generated VLIW/EPIC code range from one-eighth to one-half because a single instruction stream has limited ILP.

Software Thread Integration (STI) is a software technique which interleaves multiple threads at the machine instruction level. Integration of threads increases the number of independent instructions, allowing the compiler to generate a more efficient instruction schedule and hence faster runtime performance. We have developed techniques to use STI for converting thread level parallelism (TLP) to ILP on VLIW/EPIC architectures. By focusing on the abundant parallelism at the procedure level in the multimedia applications, we integrate parallel procedure calls, which can be seen as threads, by gathering work in the application. We rely on the programmer to identify parallel procedures, rather than rely on compiler identification. Our methods extend whole-program optimization by expanding the scope of the compiler through software thread integration and procedure cloning. It is effectively a superset of loop jamming as it allows a larger variety of threads to be jammed together.

This thesis proposes a methodology to integrate multiple threads in multimedia applications and introduce the concept of a ‘Smart RTOS’ as an execution model for utilizing integrated threads efficiently in embedded systems. We demonstrate our technique by integrating three procedures from a JPEG application at C source code level, compiling with four compilers for the Itanium EPIC architecture and measuring the performance with the on-chip performance measurement units. Experimental results show procedure speedup of up to 18% and program speedup up to 11%. Detailed performance analysis demonstrates the primary bottleneck to be the Itanium’s 16K instruction cache, which has limited room for the code expansion by STI.

SOFTWARE THREAD INTEGRATION FOR CONVERTING TLP TO ILP ON VLIW/EPIC ARCHITECTURES

by

WON SO

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh
2002

APPROVED BY:

______________________________ ______________________________

____
Chair of Advisory Committee

Biography

Won So was born in Cheonju, Korea. He got his Bachelor of Science degree in Inorganic Materials Engineering in 1997 from Seoul National University in Seoul, Korea. He worked in industry in Korea for two years as a software engineer. He began his MS study at North Carolina State University in Raleigh, NC in 2000. He obtained his Master of Science degree in Computer Engineering on Oct. 2002.

Acknowledgements

I would like to thank people who helped me with this work.

Dr. Alexander G. Dean served as my advisor and also was a mentor. He always expressed his enthusiasm for this work and gave the best advice whenever I needed it. The other committee members, Dr. Tom Conte and Dr. Eric Rotenberg, also provided me with sound advice to finish this work. Other professors at NCSU, from whom I took courses, filled up my background for this research.

My parents fully supported and encouraged me so that I could focus on my work. My uncles and aunts living in this country also helped me much with setting up a life here.

Many Korean friends here gave me a pleasant time and lots of emotions. They also helped me get through the difficult times. Student colleagues, with whom I talked about problems while studying and conducting research, were helpful.

Table of Contents

List of Tables v
List of Figures vi
1. Introduction 1
2. Integration Methods and Execution Models 4
   2.1. Identify the candidate procedures for integration 4
   2.2. Examine parallelism in the candidate procedure 4
   2.3. Evaluate the constraints for integration 6
   2.4. Performing integration 13
   2.5. Building execution models 15
3. Integration of JPEG Application and Overview of the Experiment 21
   3.1. Sample application: JPEG 21
   3.2. Integration Method 23
   3.3. Overview of the experiment and evaluation methods 25
4. Experimental Results 28
   4.1. Integrated thread 1: FDCT in CJPEG 37
   4.2. Integrated thread 2: EOB in CJPEG 38
   4.3. Integrated thread 3: IDCT in DJPEG 38
   4.4. Overall Performance Benefit 39
   4.5. Compilers and Performance 40
5. Conclusions and Future Work 42
References 43
Appendix A. Data Tables 46

List of Tables

Table 3.1. Compilers used in the experiment 27
Table 4.1. Itanium™ cycle breakdown categories 29
Table 4.2. Best Combination to invoke threads in CJPEG application 40
Table A.1. Performance data from FDCT in CJPEG 46
Table A.2. Breakdown data from FDCT in CJPEG 47
Table A.3. Performance data from EOB in CJPEG 47
Table A.4. Breakdown data from EOB in CJPEG 48
Table A.5. Performance data from IDCT in DJPEG 48
Table A.6. Breakdown data from IDCT in DJPEG 49
Table A.7. Performance data from CJPEG application 50
Table A.8. Performance data from DJPEG application 51

List of Figures

Figure 1.1. Performance benefit of STI on VLIW/EPIC machines 2
Figure 2.1. Purely independent procedure-level parallelism and strategy for STI........ 6
Figure 2.2. Call graph of DJPEG application 7
Figure 2.3. Expanded call graph of DJPEG application 9
Figure 2.4. Write buffering technique for STI of EOB in CJPEG application 11
Figure 2.5. Redesign of the control flow while performing STI of EOB in CJPEG application........ 11
Figure 2.6. Cases of integrating two identical threads and corresponding transformations for STI........ 12
Figure 2.7. Writing the integrated procedure, *jpeg_idct_islow()* in DJPEG application. 14
Figure 2.8. Two execution models: a 'direct call' and a 'Smart RTOS' 16
Figure 2.9. 'Direct call' execution models for 3 versions of JII in DJPEG. 17
Figure 2.10. Modification of the caller to call the integrated procedure (STI2/JII in DJPEG)........ 17
Figure 2.11. Running threads with the help of a 'Smart RTOS'........ 19
Figure 3.1. Algorithms of CJPEG and DJPEG........ 22
Figure 3.2. Execution time and procedure breakdown of CJPEG and DJPEG........ 22
Figure 3.3. 'Direct call' execution models for FDCT and EOB in CJPEG application........ 25
Figure 3.4. Overview of the experiment........ 26
Figure 4.1. Performance, speedup by STI, IPC, and code size of FDCT in CJPEG........ 30
Figure 4.2. Cycle and speedup breakdown of FDCT in CJPEG........ 31
Figure 4.3. Performance, speedup by STI, IPC, and code size of EOB in CJPEG........ 32
Figure 4.4. Cycle and speedup breakdown of EOB in CJPEG 33
Figure 4.5. Performance, speedup by STI, IPC, and code size of IDCT in DJPEG 34
Figure 4.6. Cycle and speedup breakdown of IDCT in DJPEG 35
Figure 4.7. Performance and speedup by STI of CJPEG application 36
Figure 4.8. Performance and speedup by STI of DJPEG application........ 36

1. Introduction

Multimedia applications are pervasive in modern systems. Still and moving images, and sounds are quite common types of information existing in the digital world. Multimedia applications generally require a significantly higher level of performance than the previous workloads. It drives digital signal processor makers to adopt high-performance architectures such as VLIW (Very-Long Instruction Word) or EPIC (Explicitly Parallel Instruction Computing). VLIW/EPIC processors improve the performance by issuing multiple independent instructions based on the decision of the compiler while limiting hardware complexity. Processors like Philips Trimedia TM100 [HA99], the Texas Instruments VelociTI architecture [TH97], the Chromatic Research Mpact1 and 2 [Purc97], the BOPS ManArray [PV00], the ST Micro STI100 CPU-DSP [DFGS00], and the Starcore SC120 from Motorola and Lucent [Half99] are the examples which implement VLIW/EPIC architecture.

The performance of VLIW/EPIC depends on the ability of the compiler to find enough instructions which can be executed in parallel. Compilers uses techniques such as loop unrolling [WS90] and software pipelining [Lam98] and forms schedules using traces [Fish81], superblocks [Hwu93], hyperblocks [MCHB92] and treegions [HBC98] to extract more independent instructions. However, typical utilization levels for compiler-generated code range from one-eight to one-half, leaving much room for improvement. [MME+97] One of the reasons is that a single instruction stream has a limited level of ILP (Instruction-Level Parallelism) because there are not enough independent instructions with in the limited size of the compiler's scheduling window. [Wall91] In order to overcome this limit, techniques to exploit more parallelism by extracting multiple threads out of a single program were introduced. This parallelism is called Thread-Level Parallelism (TLP) because the independent instructions are located more distant. Many architectural designs are proposed to exploit TLP. Those include Simultaneous Multithreading (SMT) [TEL95], Multiscalar [SBV95], Dynamic Multithreading (DMT) [AD98], Welding [OCS01] and Superthreading. [THA+99]

Software Thread Integration (STI) is a software technique which interleaves multiple threads at machine instruction level. [DS98] Previous research proposed a method by which a thread with real-time constraints can be integrated with another thread and introduced the post-pass compiler *Thrint*, which automates integration processes, enabling Hardware-to-Software Migration (HSM) by moving some dedicated hardware functions to software without real-time OS overhead. [Dean00] [Dean02] The other benefit of STI is that the integration of threads increases the number of independent instructions, allowing the compiler to generate a more efficient instruction schedule and hence faster runtime performance. STI can convert abundant thread-level-parallelism (TLP) in multimedia applications to instruction-level-parallelism (ILP) to achieve implicit multithreading in a single-threaded machine with no architectural support. Figure 1.1 illustrates how the integrated thread can achieve performance improvement. In this research, we leverage this benefit for improving the performance of multimedia applications running on VLIW/EPIC machine. We integrate parallel procedure calls, which can be seen as parallel threads, to improve run-time performance by gathering work in an application and executing it efficiently. We rely on the programmer to identify parallel procedures, rather than rely on compiler identification. STI fuses entire procedures together, removing the loop-boundary constraint of loop jamming or fusion. [Ersh66] It uses code replication and other transformation as needed to reconcile control-flow differences.

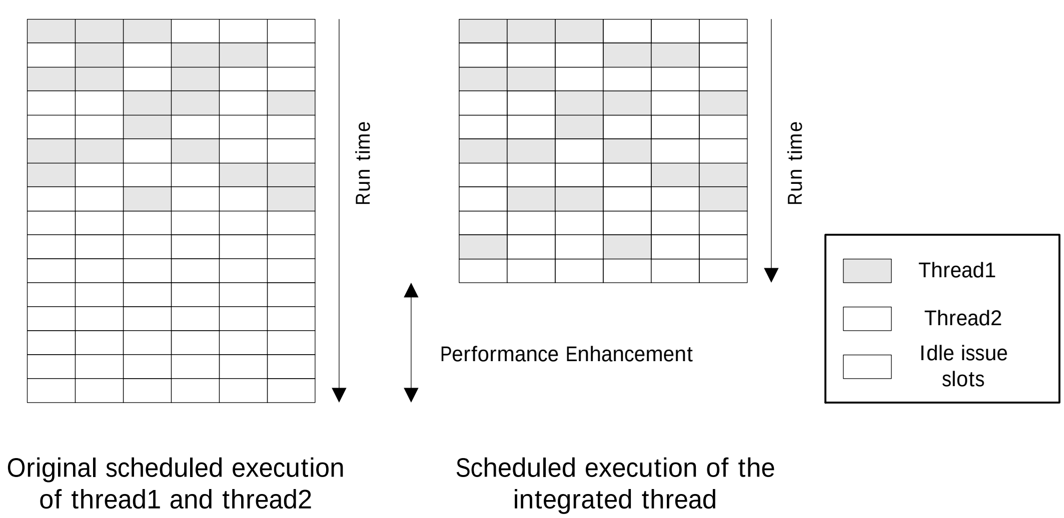


Figure 1.1. Performance benefit of STI on VLIW/EPIC machines

Whole-program (interprocedural) optimization methods extend the compiler's scope beyond the call boundaries defined by the programmer to potentially encompass the entire program. The methods include interprocedural data-flow analysis, procedure inlining and procedure cloning. Procedure cloning consists of creating multiple versions of an individual procedure based upon similar call parameters or profiling information. Each version can then be optimized as needed, with improved data-flow analysis precision resulting in better interprocedural analysis. The call sites are then modified to call the appropriately optimized version of the procedure. Procedure cloning is an alternative to inlining; a single optimized clone handles similar calls, reducing code expansion. Cooper and Hall [CHK93] [Hall91] used procedure cloning to enable improved interprocedural constant propagation analysis in the matrix300 from SPEC 89. Selective specialization for object-oriented languages corresponds to procedure cloning. Dean at el. [DCG94] use static analysis and profile data to select procedures to specialize.

STI proposes a way to provide procedure clones by integrating parallel procedure calls exploiting TLP in an application. The novelty is in providing clone procedures that do the work of two or three procedures concurrently ("conjoined clones") through software thread integration. This enables work to be gathered for efficient execution by the clones. This approach is also different from other proposed architectural models to exploit TLP in an application, as it does not require any architectural support. In this thesis, we present the methods to clone and integrate procedures in multimedia applications to improve the run-time performance and introduce the concept of 'Smart RTOS' as a model for utilizing integrated threads efficiently in embedded systems. Then we present the results which show how much performance benefit we can achieve using STI and what limits it with the sample application. We examine JPEG on an Itanium™ processor compiling it with four different compilers. (gcc, ORCC, Pro64, and Intel C++)

The organization of this thesis is as follows. Chapter 2 presents the general methodology from finding threads for integration to performing integration and execution models for integrated threads. Chapter 3 describes how we performed the integration for JPEG application and the overview of the experiment. Chapter 4 presents and analyzes the experimental results. The thesis ends with the conclusions in Chapter 5. Appendix A. includes data tables from the experiment.

2. Integration Methods and Execution Models

Planning and performing STI requires several steps. We choose the candidate procedures from the application and examine the applicability of STI. Then we perform integration to create the integrated versions of target procedures and use those in the application with a certain model of execution. These five steps are presented in detail below.

2.1. Identify the candidate procedures for integration

The first stage of integration is to choose the candidate procedures for integration from the application. Most programs are composed of many procedures, which are called and call each other. However, programs tend to spend most of their execution time for running a few procedures. The best tool for identifying those is profiling, which is supported by most compilers (e.g. *gprof* in gnu tools). It is quite clear that improving the performance of the program is best accomplished by improving the procedure take most of the program's time. Profiling shows how much percentage of time is spent on each procedure in an application. The procedure which consumes more time than any other else will be the first candidate for integration.

For multimedia applications, those procedures usually include compute intensive code fragments, which most DSP benchmark suites call *DSP-Kernels* such as filter operations (FIR/IIR), and frequency-time transformations. (FFT, DCT) [ZMSM94] Those routines have more loop-intensive structures and handle larger data sets, which requires more memory bandwidth then normal applications. [FWL99]

2.2. Examine parallelism in the candidate procedure

The second step is to examine parallelism in the candidate procedure because integration requires parallel execution of procedures. There exist various levels of parallelism in the applications from instruction to loop and procedure levels based on the distance of the independent instructions. Instruction level parallelism (ILP) is the execution of independent instructions within limited size of instruction window or somewhat bigger scope. ILP architectures and compilers are designed to exploit this level of parallelism. There is another level of parallelism called thread level parallelism (TLP), where independent instructions are far more distant than those in ILP. This parallelism cannot be found and used easily by ILP-centric hardware mechanisms or compilers because independent instructions are too distant to be detected with a small instruction window. [MOAV99]

The method proposed here is a software technique to use STI for converting procedure-level parallelism to ILP. Though there are other levels of TLP in the applications, we only focus on this type. Multimedia applications tend to spend most of their execution time to run compute intensive routines. Those routines handle large sets of data located in memory. They are iteratively called in the applications and the input and output sets for each call are independent of each other. This characteristic is distinct because media applications process their large data by separation into blocks. For example, FDCT/IDCT (forward/inverse discrete cosine transform), one of the most common processes in image applications, handles a 8x8 independent block of pixels; these procedures are called many times in the applications like JPEG and MPEG. We use this *purely independent procedure-level data parallelism:* 1) Each procedure call handles its own data set, input and output. 2) Those data sets are purely independent of each other requiring no synchronizations between calls. Our strategy for STI is to rewrite the target procedure to handle multiple sets of data at once. An integrated procedure joins multiple independent sets of instruction streams and can offer more ILP than the original procedure. Figure 2.1 shows the existing parallel procedure calls and strategy for integration. Parallel procedure calls can be seen as parallel threads.

Detecting this parallelism is not an easy task; much work has been done to automatically extract multiple threads out of a single program for execution on processors with multiple instruction streams. For example, the PARADIGM compiler automatically extracts abundant loop level parallelism in the scientific programs and distributes program data for distributed-memory multicomputers. [BCG+95] The Multiscalar approach uses a compiler to extract tasks from a sequential program for concurrent speculative execution by partitioning Control Flow Graphs (CFG). [SBV95] The Pedigree compiler reveals the parallelism with Program Dependence Graphs (PDG) and performs automatic partitioning and scheduling for XIMD processors. [Newb97] Nevertheless, identifying procedure-level parallelism requires interprocedural data flow analysis, which is still quite difficult in the general case. [Ryde82] We are not trying to solve this problem. Instead, we assume that application developers will extract parallel threads whether automatically or manually. We only present a method to execute the parallel procedures more efficiently on a single instruction stream processor. In this research, we examine parallelism in a specific procedure based on algorithmic understanding of the application and observation of the program's behavior.

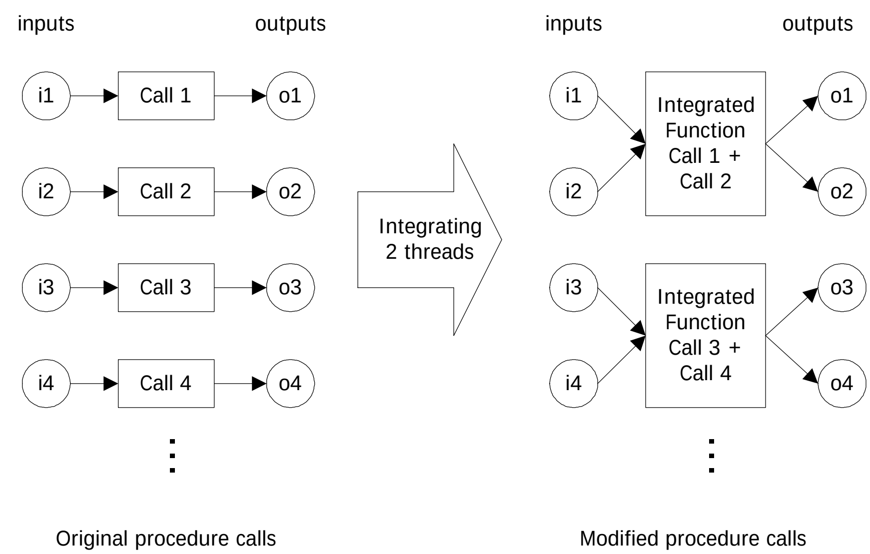


Figure 2.1. Purely independent procedure-level parallelism and strategy for STI

2.3. Evaluate the constraints for integration

The third step is to evaluate the potential difficulties for integration. Since integration needs code transformation in procedure level, it depends on the particular code structure rather than the general characteristics of the application. Even though procedure calls are independent, the code structure of the program occasionally makes the code transform hard.

The first constraint comes from the data tractability for moving calls. To call the integrated procedure in the application, calls must be advanced or delayed to the position where another procedure is called. Though the procedure calls are independent, moving calls for integration causes another data flow issue: the change of def-use chain for inputs and outputs. It requires complicated interprocedural analysis. In this research, we limit that complexity by choosing the procedures which give a reasonable scope to examine it manually. When the calls are too apart through many procedure calls, it is hard to track all data sets and possible changes. It can be identified by constructing the expanded call graph, which shows the hierarchy and behavior of the procedure call pattern, using a profiler and a debugger.

Call graph profile listing can be used to construct the call graph of the program, which shows the hierarchy of procedure calls. [GKM82] Figure 2.2 shows the simplified call graph of DJPEG application, which includes only candidate functions from DJPEG application. Rectangles represent procedures. Callers and callees are connected with arrows, the number on which represents number of calls from callers to callees. Since the call graph shows only the static behavior of procedure calls, the pattern of multiple call instances should be also obtained with the help of a debugger like *gdb*. By instantiating all call instances from the call graph, the expanded call graph is constructed as we see in Figure 2.3.

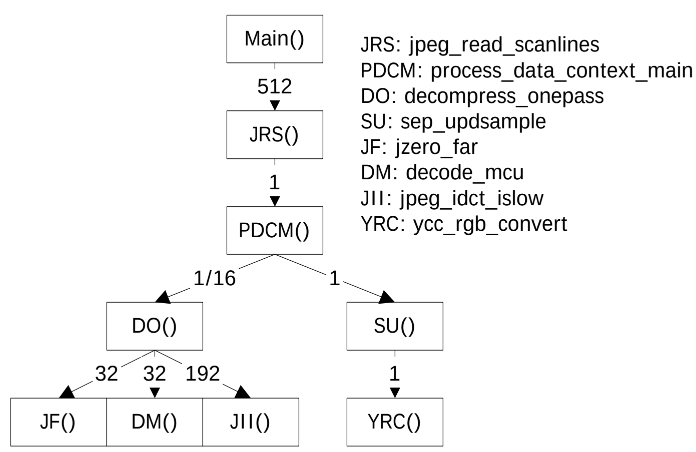


Figure 2.2. Call graph of DJPEG application

Data tractability for integrating multiple procedure calls can be examined by observing the expanded call graph. If multiple instances of calls are activated with nested intervening calls, it is not easy to make sure that all the inputs and outputs are not changed through those procedure calls. For example, the multiple instances of the candidate function YRC in DJPEG (in Figure 2.3) are activated by an ancestor procedure Main with three intervening procedure calls, JRS, PDCM and SU. When we defer the 1st YRC call to the 2nd one for parallel execution, it is hard to make sure that inputs and outputs of that procedure call are not changed through four procedure calls. On the other hand, the multiple calls of the candidate procedure JII are activated by a function DO, which is located just above JII. Therefore, JII gives a reasonable code scope to track data for integration while YRC does not. (The dotted rectangles in Figure 2.3 represent the respective scopes for two respective procedures.) Too big code scope for tracking data would require interprocedural data flow analysis, which is considered to be too complicated. We focus only on procedures like JII, in which calls are adjacent and activated by one parent procedure call. It gives a reasonable scope to modify the source code.

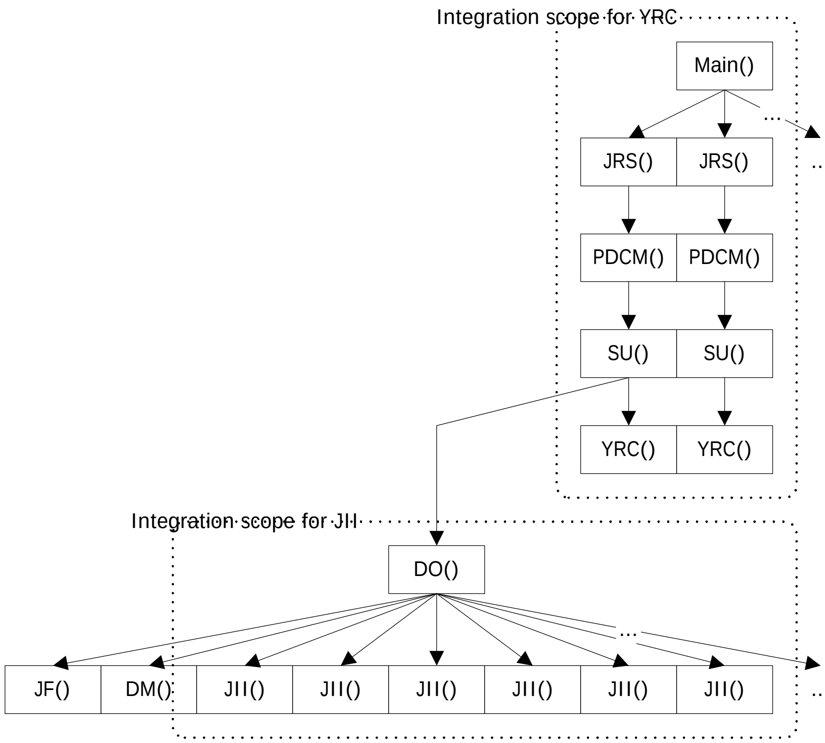


Figure 2.3. Expanded call graph of DJPEG application

The second constraint comes from the reuse of the buffer. Some buffers need to be duplicated to allow concurrent execution of separate procedures. Since multimedia applications use large data sets, the input and output buffers are often reused in order to save memory space. If the buffer is composed of multiple levels with variable sizes and forces outputs to be written in a specific order, a buffering technique need to be be used for integration. For example, Huffman encoding, which is used for encoding DCT coefficients in CJPEG, is a scheme in which encoded codeword has a variable bit size. The buffer to write codewords is composed of multiple levels and the procedure, *emit_bits*, is in charge of writing a codeword to the buffer sequentially in bit level. It writes codewords to another lower level of buffer when its buffer is full. The original procedure, *encode_one_block (EOB)*, calls that procedure, *emit_bits* whenever it computes a codeword. We normally cannot interleave all the codes in that procedure because the written order also would be interleaved by performing integration and mix up the codewords. To solve this problem, writing to the original target buffer must be delayed and buffered to the temporary buffer and all codewords must be written to it after it finishes decoding. Figure 2.4 illustrates the way to maintain the write order and Figure 2.5 shows how the control flow of the integrated procedure should be redesigned. Therefore, buffering makes the written order same as the original procedure but requires an additional local buffer, which could increase data accesses and register pressure.

On the other hand, integration is impossible if there is this kind of constraint in read side. The procedure which performs Huffman decoding, *decode_mcu (DM)* in DJPEG is the counter part of EOB in CJPEG. Decoding cannot be parallelized because the position of the buffer, from which the codewords for the next coefficients block are read, can be identified only after it finishes decoding the previous block.

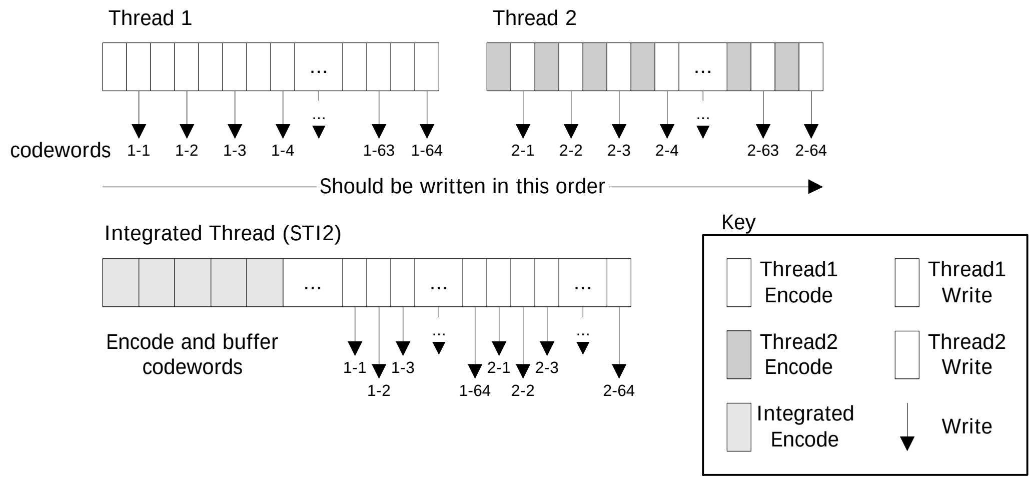


Figure 2.4. Write buffering technique for STI of EOB in CJPEG application

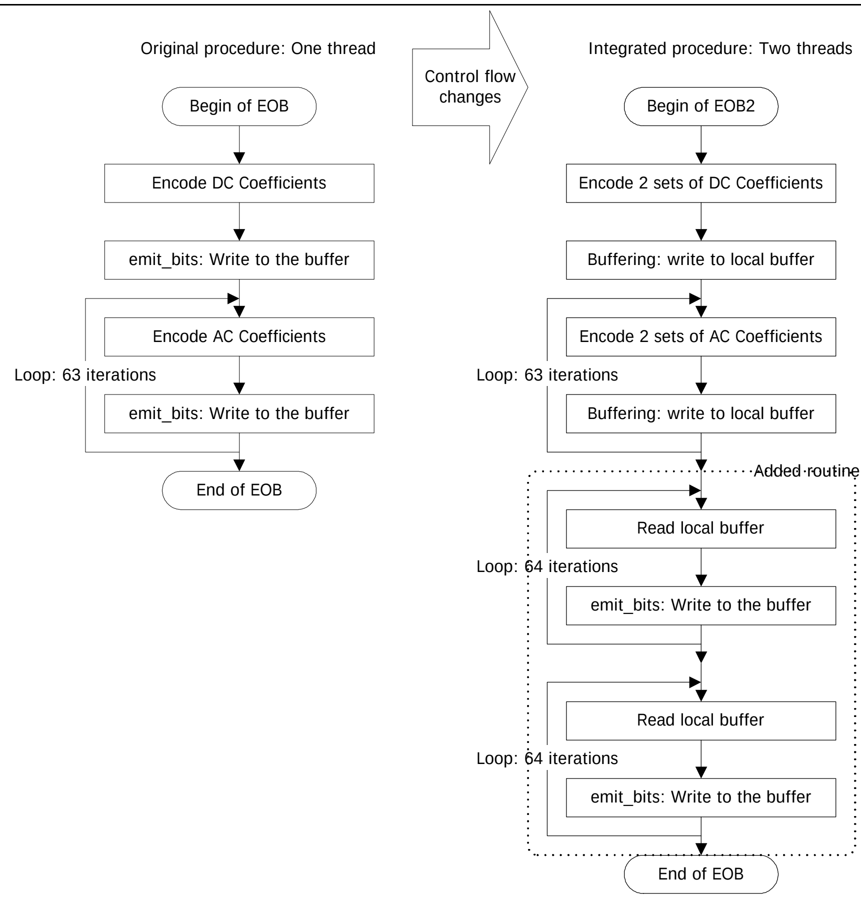


Figure 2.5. Redesign of the control flow while performing STI of EOB in CJPEG application

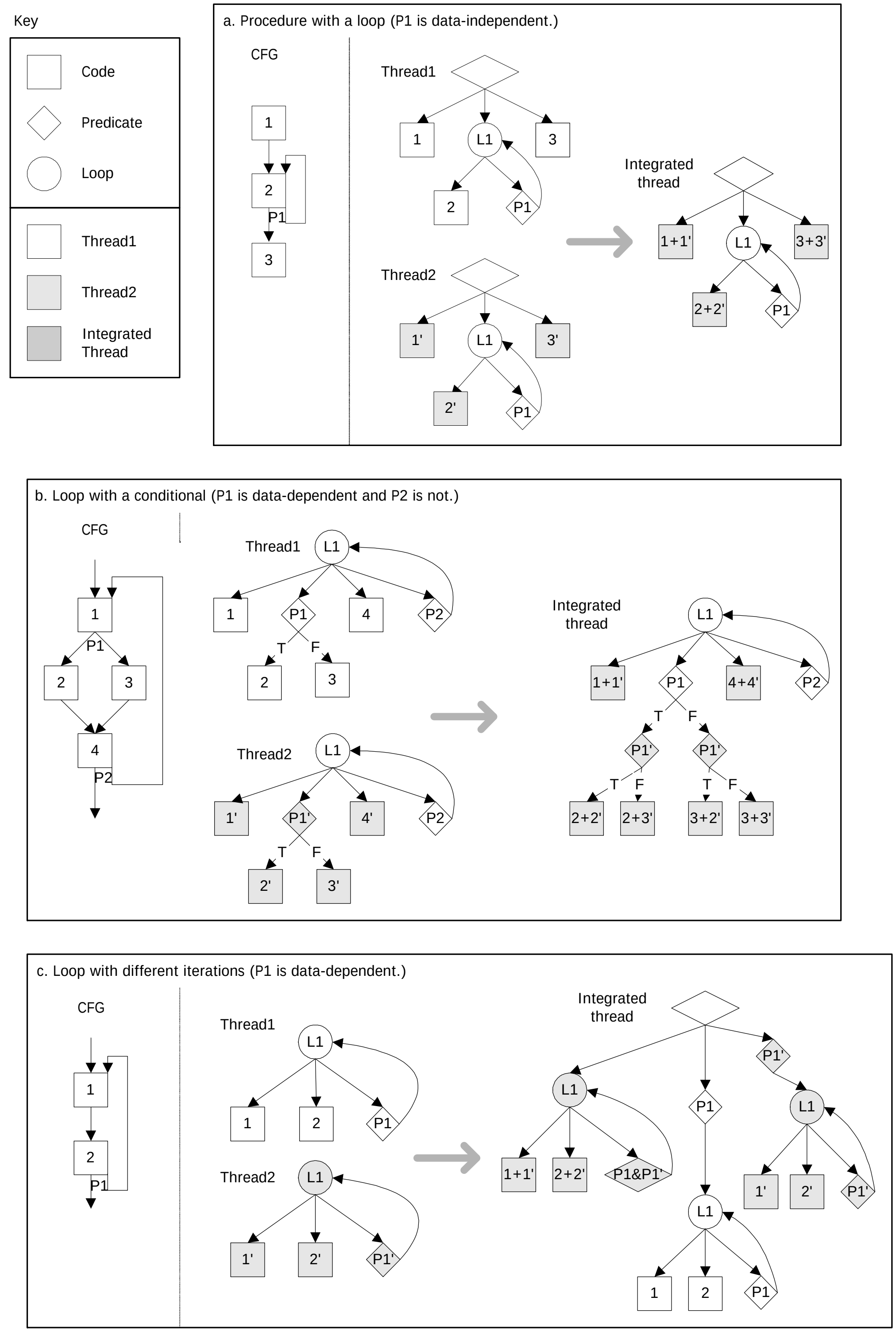


Figure 2.6. Cases of integrating two identical threads and corresponding transformations for STI

2.4. Performing integration

When the candidate procedure has the desired parallelism and few constraints for the code transformations, we can determine which procedures to integrate. We can also determine how many procedure calls can be integrated from the expanded call graph. Since we only consider integrating adjacent calls from one parent procedure, how many threads can be integrated is limited by number of adjacent calls. For example, integration of calls from 2 to 6 of JII is possible. However, integrating too many procedure calls is not desirable because it increases the code size excessively.

Many cases and corresponding code transform techniques for STI have been already demonstrated by previous work. [DS98] [Dean00] [Dean02] STI uses the control dependence graph (CDG, a subset of the program dependence graph) [FOW87] to represent the structure of the program, which simplifies analysis and transformation. STI interleaves multiple procedures, with each implementing part of a thread. For consistency with previous work we refer to the separate copies of the procedure to be integrated as threads. Integration of identical threads is a simple case. Figure 2.6 demonstrates the cases and corresponding code transformation, which can happen integrating the same threads which handle different data sets. This transformation can be applied repeatedly and hierarchically, enabling code motion into a variety of nested control structures. Integration of basic blocks involves fusing two blocks. (case a) To move code into a conditional, it is replicated into each case. (case b) Code is moved into a loop with guarding or splitting. Finally, loops are moved into other loops through combinations of loop fusion, peeling and splitting. (case c) These transformations can be seen as a superset of loop jamming or fusion. They jam not only loops but also all code (including loops and conditionals) from multiple procedures or threads, greatly increasing its domain.

Code transformation can be done in two different levels: assembly or high-level-language (HLL) level. Assembly level integration is preferred generally because it schedules instructions with fine-grain concurrency. It especially works efficiently with processors like superscalars which have dynamic scheduling capabilities because find-grained independent instructions are more likely to be executed simultaneously. However, VLIW/EPIC not only require interleaving instructions but also meeting scheduling constraints like resource distribution into appropriate issue slots. Though HLL level integration tends to result in coarse-grain concurrency, it can be chosen as an alternative because it can reduce a programmer's effort when there is no automatic tool for code transformation. Furthermore, if there is a good compiler support for VLIW or a large instruction window size in superscalar, it could offer fine-grain concurrency. There is no general answer for which level integration is better. It primarily depends on the capabilities of the tools and compilers.

Whether the integration is done at either assembly or HLL level, it requires two steps. The first is to duplicate and interleave the code (instructions). The second is to rename and allocate new local variables and procedure parameters (registers) for the duplicated code. Figure 2.7 demonstrates how the integrated procedure can be written while performing integration of two threads at the C level. The second step is quite straightforward during HLL level integration because the compiler takes care of allocating registers. Not all local variables (registers) are duplicated because there could be some variables shared by the threads.

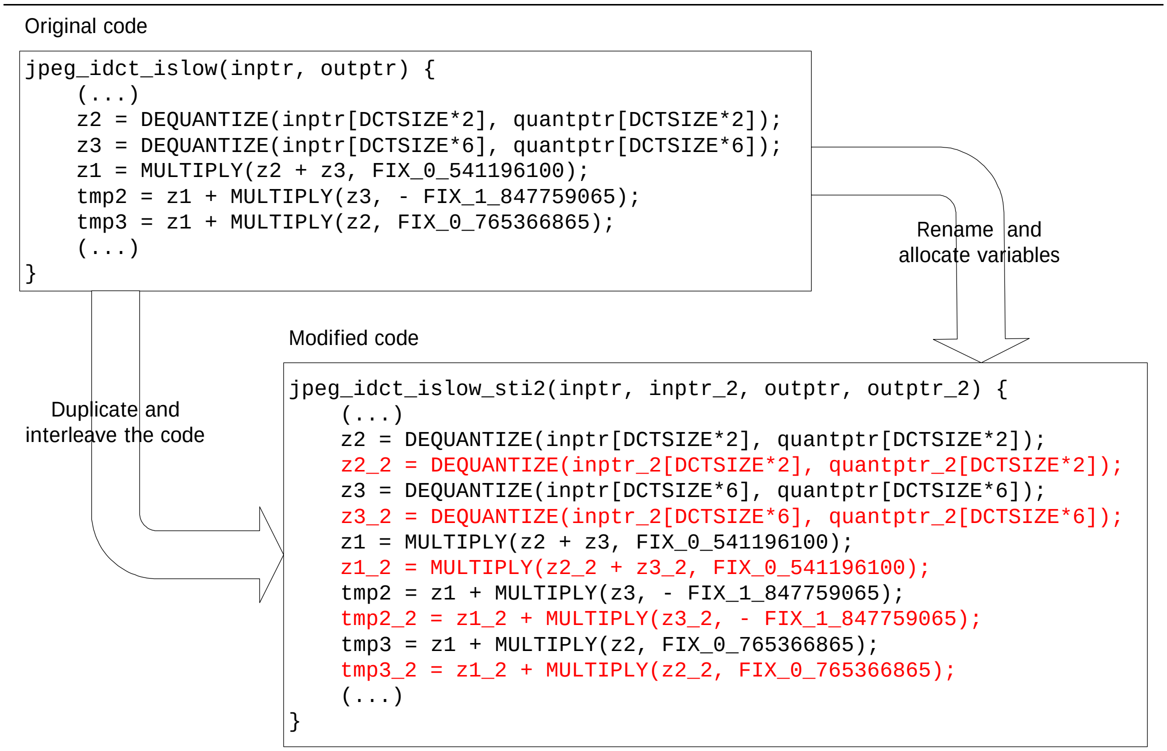


Figure 2.7. Writing the integrated procedure, *jpeg_idct_islow()* in DJPEG application.

There are two expected side effects from these code transforms. One is a code size increase (code expansion), and the other is a register pressure increase. The best-case code size increase is approximately proportional to the number of integrated threads. If there is a conditional which depends on the data, the code size increase is exponential to the number of threads and conditional nesting depth. Code size increase has a significant impact on performance if it exceeds a certain threshold, which determined by instruction cache size and levels. The number of registers also increases linearly with the number of integrated approximately. It also has a negative influence on the performance if it causes register spills due to insufficient physical registers. Control flow does not affect the register pressure.

Negative impact on performance from both exceeds the benefit from STI if the code size is excessive. Performance degradation from I-cache misses is especially significant if the code size is larger than L1 I-cache size. Optimizations, which increase the code size significantly (such as loop unrolling), should be disabled before integration in such cases. Though code size of the integrated procedure cannot be measured before integration, the expected code size can be estimated based on that of that of the original procedure. If it exceeds the L1 I-cache size much, we cannot expect performance improvement by integration.

2.5. Building execution models

There are two approaches to use the integrated threads in the applications. Figure 2.8 compares these two approaches. The first approach is to modify the application to call integrated threads directly by replacing original procedure calls with integrated procedure calls. (Static approach: 'direct call') This is appropriate for a simple system where there is only one application running and we are interested only in improving its performance. The second is to develop a 'Smart RTOS' to select the efficient version of the procedure dynamically. (Dynamic approach: 'Smart RTOS') This is useful for a complicated system where there are multiple applications running on the system and the environment changes frequently.

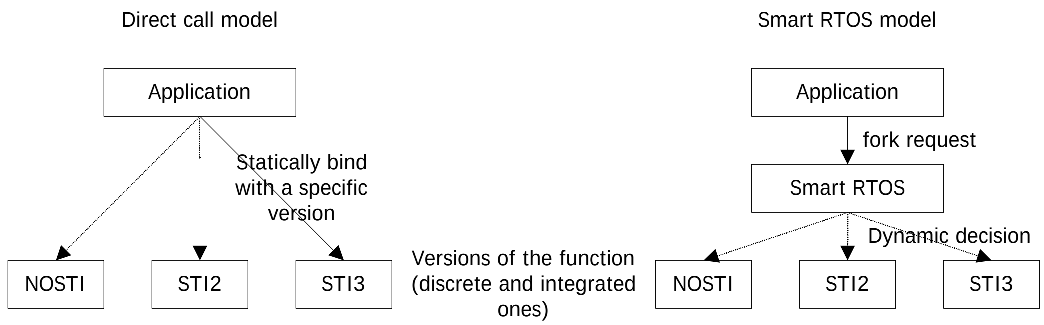


Figure 2.8. Two execution models: a 'direct call' and a 'Smart RTOS'

(1) Direct call

Direct call is the simple way to use the integrated threads. After writing the integrated versions of the target procedures, we include those in the original source code and modify the caller procedure to call the specific version of the callee procedure every time. Figure 2.9 illustrates the simple execution models to run integrated versions of JII procedure in the DJPEG application.

Typically the caller procedure is organized to call the target procedure a certain number of times with the form of a loop. Since the integrated procedure handles multiple calls at once, the caller must delay the calls and store the procedure parameters until it has data for the multiple calls. Then it calls the integrated procedure with multiple sets of parameters. Some local variables for storing parameters for delayed calls and for organizing the control flow are allocated to the caller and control flow becomes slightly more complicated than before. As a result, some overhead is unavoidable from register pressure and branch misprediction. Figure 2.10 shows how the caller was modified for integrated version of JII in DJPEG application. Not all caller procedure are as simple as this example but some variations could be easily taken care of by monitoring changes of parameters.

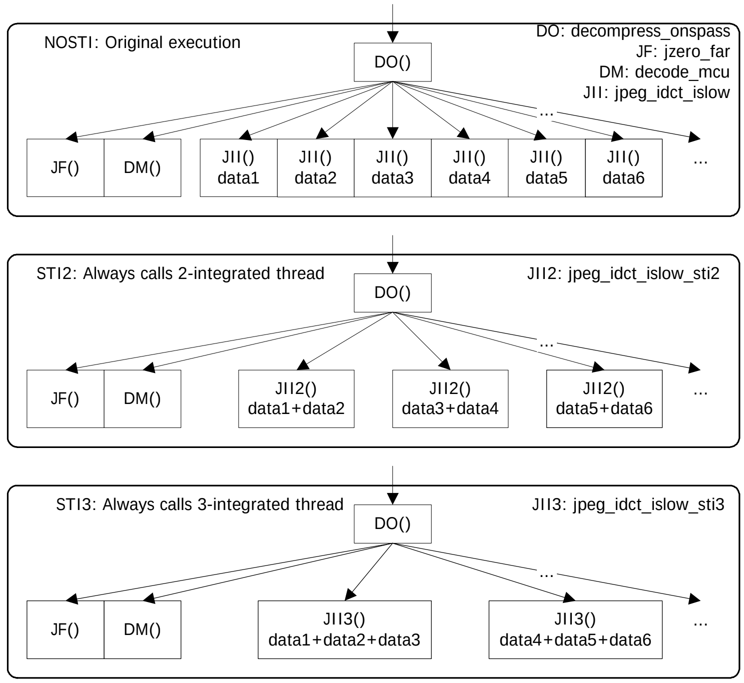


Figure 2.9. 'Direct call' execution models for 3 versions of JII in DJPEG.

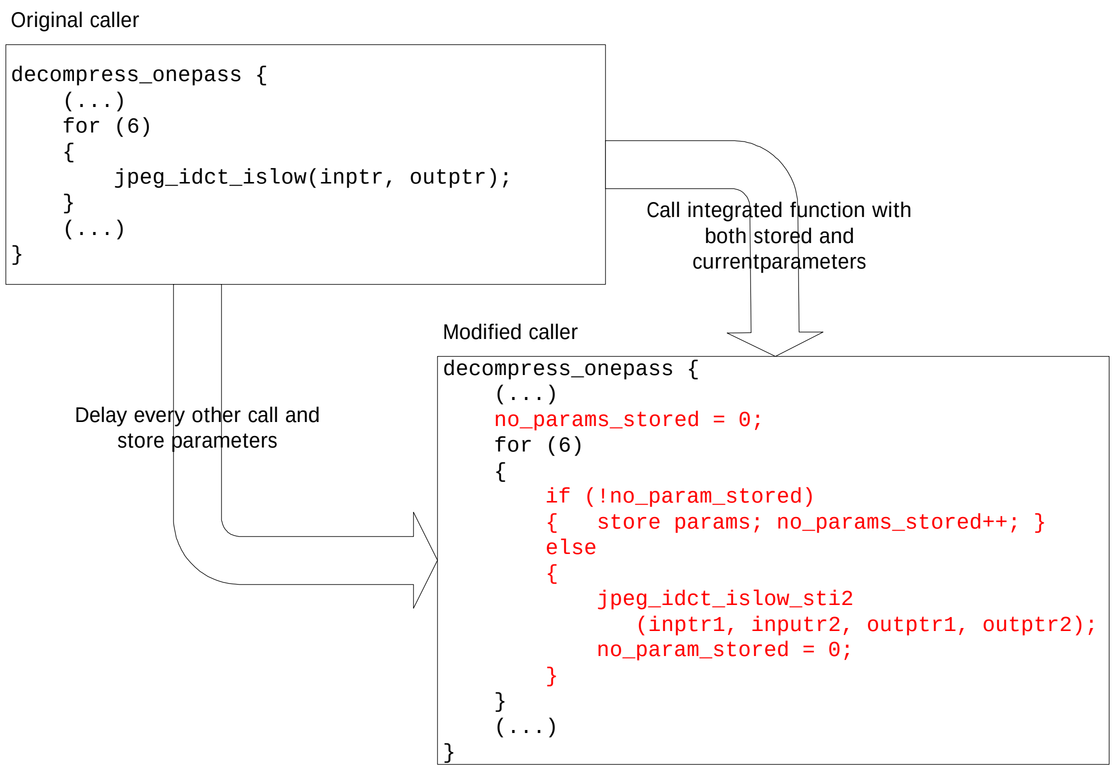


Figure 2.10. Modification of the caller to call the integrated procedure (STI2/JII in DJPEG)

The application can be optimized using feedback based on the performance of each version of the procedure. We measure the performance of various versions of the application (or procedures), varying the level of integration in the procedures. Optimization is achieved by selecting the most efficient version. This grows more important if we have more than one procedure with various versions in the application. For example, we have three versions – original, 2-thread integrated, 3-thread integrated – of FDCT and Encode (Huffman encoding) in CJPEG application. From the nine possible ways to invoke those two threads, the best combination can be obtained by choosing most efficient version of respective FDCT and Encode. This leads to optimization of the application.

(2) Smart RTOS

A ‘Smart RTOS’ can be introduced to run the integrated threads efficiently. If we succeed in writing multiple versions of the target procedures, we create a library of integrated and discrete versions of procedures. We modify the program to replace the original procedure calls with RTOS *thread-forking requests*. Then we enhance the RTOS to interpret thread-forking requests. It can either choose the best version of a thread to execute from library of multiple versions and invoke it immediately or else defer its execution, waiting for another thread request which would enable the execution of an integrated version. The goal of the selection is to choose a version of the thread which has been integrated with one or more other threads and is most efficient.

Figure 2.11 illustrates how the RTOS uses integrated threads. Thread A, B, and C in applications (a single application or different applications or multiple instances of the same application) send thread-forking requests to the Smart RTOS. The Smart RTOS saves those requests into their respective queues for different threads, and the scheduler in the Smart RTOS determines whether to execute immediately or defer execution. When it decides to execute a specific thread, it also determines which version of thread is the most efficient to run at that moment considering combinations of all pending threads in the queues. Then it invokes a specific version of thread from the thread library.

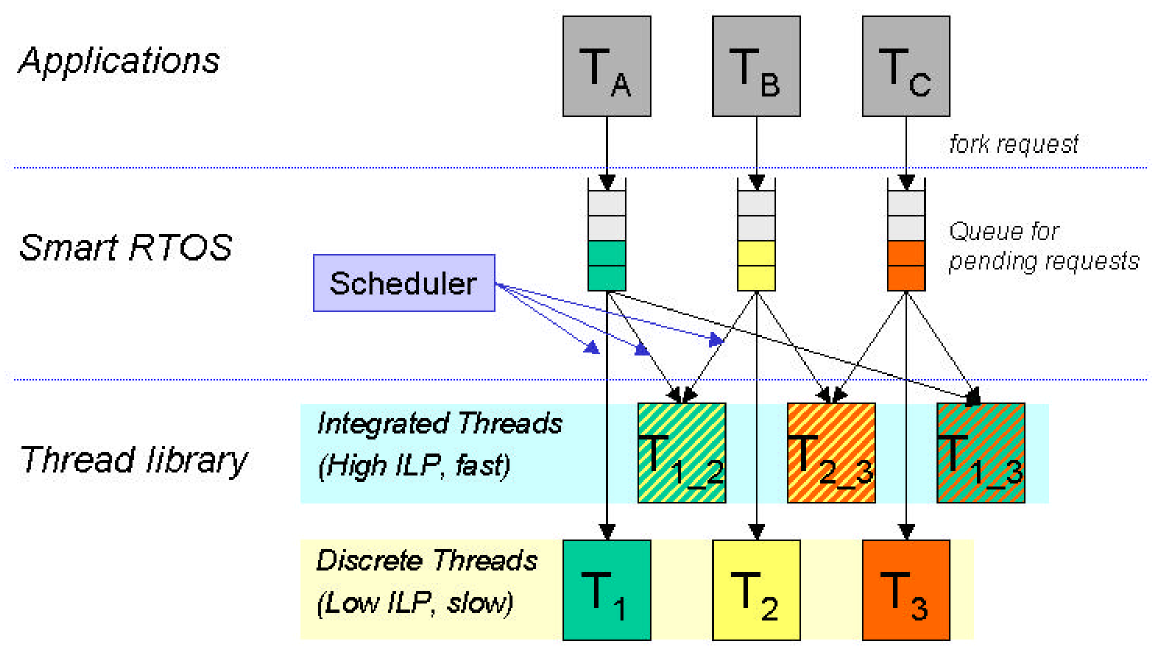


Figure 2.11. Running threads with the help of a 'Smart RTOS'

A Smart RTOS gives much flexibility to take advantage of STI. First, it enables exploiting the parallelism of multiple applications. Just as a single instruction stream has an ILP limit, a single application has limited TLP itself. In a complicated system which runs many applications, there is more TLP which can be used for STI. For example, a cell phone base station processes data for multiple voice channels for many users simultaneously. We can use STI technique for the computation kernels for encoding and decoding voice and achieve performance improvements. Second, a Smart RTOS can adaptively offer optimal executions for changing workload of a system because it is based on dynamic decisions. If requests for a certain thread have reduced significantly at some moment, it can monitor that change and try not to wait for that request. Instead it can try to invoke a discrete version of that thread immediately, which gives quicker execution. Third, the goal of scheduling can be extended. Now we are focusing only on improving the performance of applications but most embedded systems require meeting deadlines for some processes. An RTOS can maintain the information about the execution time of various versions of the threads and invoke appropriate versions to meet the deadline of the specific thread.

However, there are some unavoidable consequences in this approach. First, the RTOS will introduce some overhead for the execution depending on the complexity of the scheduler algorithm. If the algorithm is too computationally expensive, it will exceed the benefits of integrated threads. Second, kernel procedures of the applications should be carefully designed to share requests and applications should be written in that way. The same procedures from different applications should have the same procedural interface in order to share requests for that thread. As this thesis is an exploration of the potential performance impact of integrating threads, the Smart RTOS concept is not examined further here and is left for future work.

3. Integration of JPEG Application and Overview of the Experiment

We chose the JPEG application as an example of the multimedia workload. We performed STI for JPEG application as presented in Chapter 2. Three target procedures were identified and integrated manually at the C source code level and executed in the application with a ‘direct call’ execution model on an Itanium^{TM} machine. The objective of the experiment is to build up the general approach to perform STI and to examine performance benefits and bottlenecks. We do not use a ‘Smart RTOS’ model for this experiment because we focus only on measuring and analyzing the performance of integrated threads and utilization of the processor.

3.1. Sample application: JPEG

JPEG is a standard image compression algorithm and one of the most frequently used one in multimedia applications. It is contained in the MediaBench benchmark suite, a popular benchmark for representing multimedia workloads. [LPM97] Source code was obtained from Independent JPEG Group. Every configuration setting was maintained from the original one but the input file. We used 512x512x24bit lena.ppm, which is a standard image for image compression research.

JPEG is composed of two application: DJPEG (Decompress JPEG) and CJPEG (Compress JPEG). Understanding the algorithm of the application helps find existing parallelism in the application. The basic compress and decompress algorithms are presented in Figure 3.1. [Hals01] [Wall92]

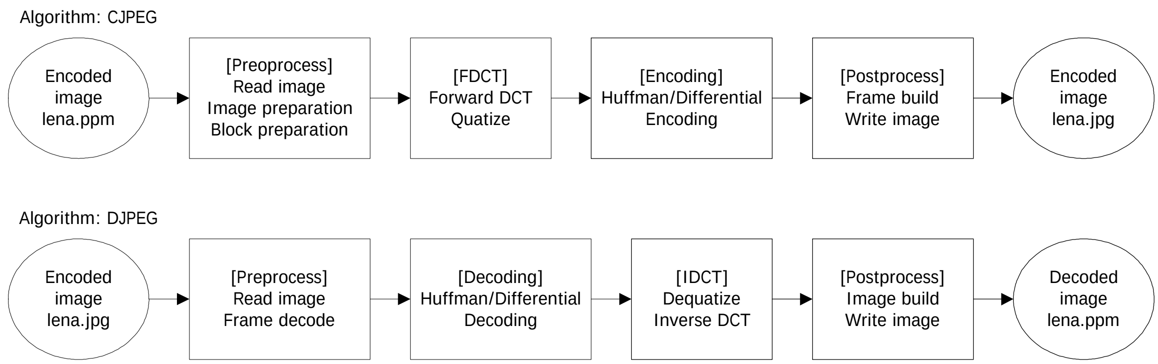
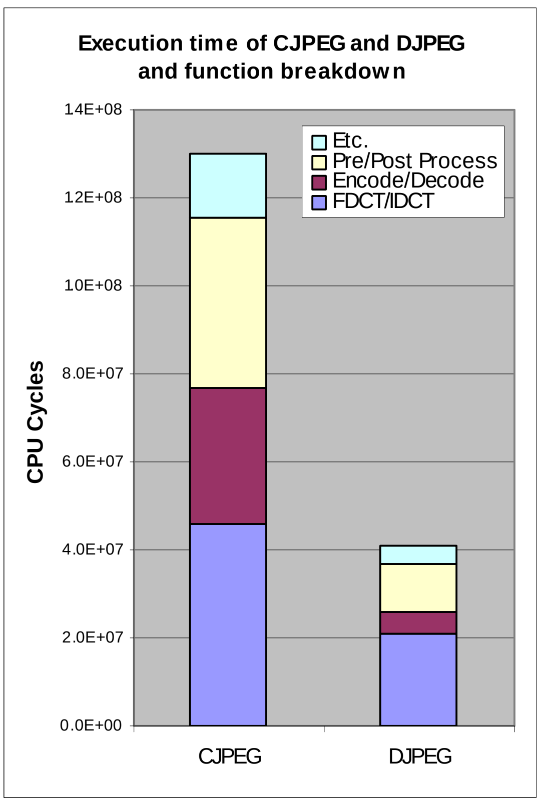


Figure 3.1. Algorithms of CJPEG and DJPEG



CJPEG	
Time(%)	Procedure
23.86	encode_one_block
20.45	preload_image
18.18	forward_DCT
17.05	jpeg_fdct_islow
9.09	rgb_ycc_convert
4.55	get_24bit_row
4.55	h2v2_downsample
2.27	__divdi3
DJPEG	
Time(%)	Procedure
51.02	jpeg_idct_islow
26.53	ycc_rgb_convert
12.24	decode_mcu
8.16	h2v2_fancy_upsample
2.04	decompress_onepass
Categories	Encode/Decode
	Pre/Post Process
	FDCT/IDCT

Figure 3.2. Execution time and procedure breakdown of CJPEG and DJPEG

3.2. Integration Method

Figure 3.2 shows the execution time and procedure breakdown of CPU cycles from gprof compiled by GCC with –O2 optimization. DJPEG uses most of its execution time to perform IDCT (Inverse Discrete Cosine Transform). CJPEG also uses significant amounts of time to perform FDCT (Forward DCT) and Encoding (Huffman Encoding).

FDCT/IDCT is a common technique for compressing and decompressing an image. The existing parallelism is that one procedure call performs FDCT/IDCT for an 8x8 pixel macro block; the input and output data of every procedure call are independent. Similarly Encode/Decode processes a block of data per procedure call and has the same level of parallelism as FDCT/IDCT has.

We perform the integration of two and three threads for IDCT, *jpeg_idct_islow (JII)*, in DJPEG and FDCT, *forward_DCT (FD)* and *jpeg_fdct_islow (JFI)*, Encode, *encode_one_block (EOB)*, in CJPEG. Decode, *decode_mcu (DM)*, in DJPEG cannot be parallelized because of the data dependencies between buffer positions of the blocks. We do not perform the integration for other procedures, *rgb_ycc_convert (RYC)* and *ycc_rgb_convert (YRC)* because their calls are far apart from each other through three ancestor procedures, which would cause too many changes of the programs. (See Figure 2.3.)

Code transformation is done at the C source level using the techniques just presented. IDCT is composed of two loops with identical control flow and a conditional which depends on the input data. The control flow of the integrated procedure is structured to handle all possible execution paths. (case b in Figure 2.6) The control flow of Encode in CJPEG is also similar as it has a loop with a data-dependant predicate. The previously mentioned buffering technique is applied to maintain the write order of codewords during integration of EOB. FDCT is composed of two procedures, FD and JFI, for which are integrated respectively. Even though there is a nest call from FD to JFI, the control flow is quite straightforward as it is an extension of a procedure with a loop. (case a in Figure 2.6) In this case, intermediate function FD is modified to call JFI properly so that it can handle the right data sets.

We invoke the integrated threads by the 'direct call' execution model. Three different versions of caller procedures for the respective versions are written and included in the original source code, and are compiled to different versions of the application with conditional compile flags. Then we measure the performance of the target procedures with those versions. Finally, we build the best-optimized version of the application by selecting the version with the best performance.

Since this experiment is focused on the performance improvement from STI, it does not include any detailed model for the 'Smart RTOS'. However, 'direct call' execution model just represents the simplest model of a 'Smart RTOS', which chooses the same version of the thread at the every thread-forking request without any RTOS overhead. Original execution and two implemented models (STI2 and STI3) for IDCT in DJPEG are shown in Figure 2.8. Figure 3.3 illustrates models for FDCT and EOB in CJPEG. The call pattern of JFI is not as straightforward as JII is in DJPEG and EOB in CJPEG because the procedure FD, which is responsible for partial function for FDCT, is located between Compress_data and JFI. The procedure FD is modified in order that Compress_data call the respective versions of FDCT.

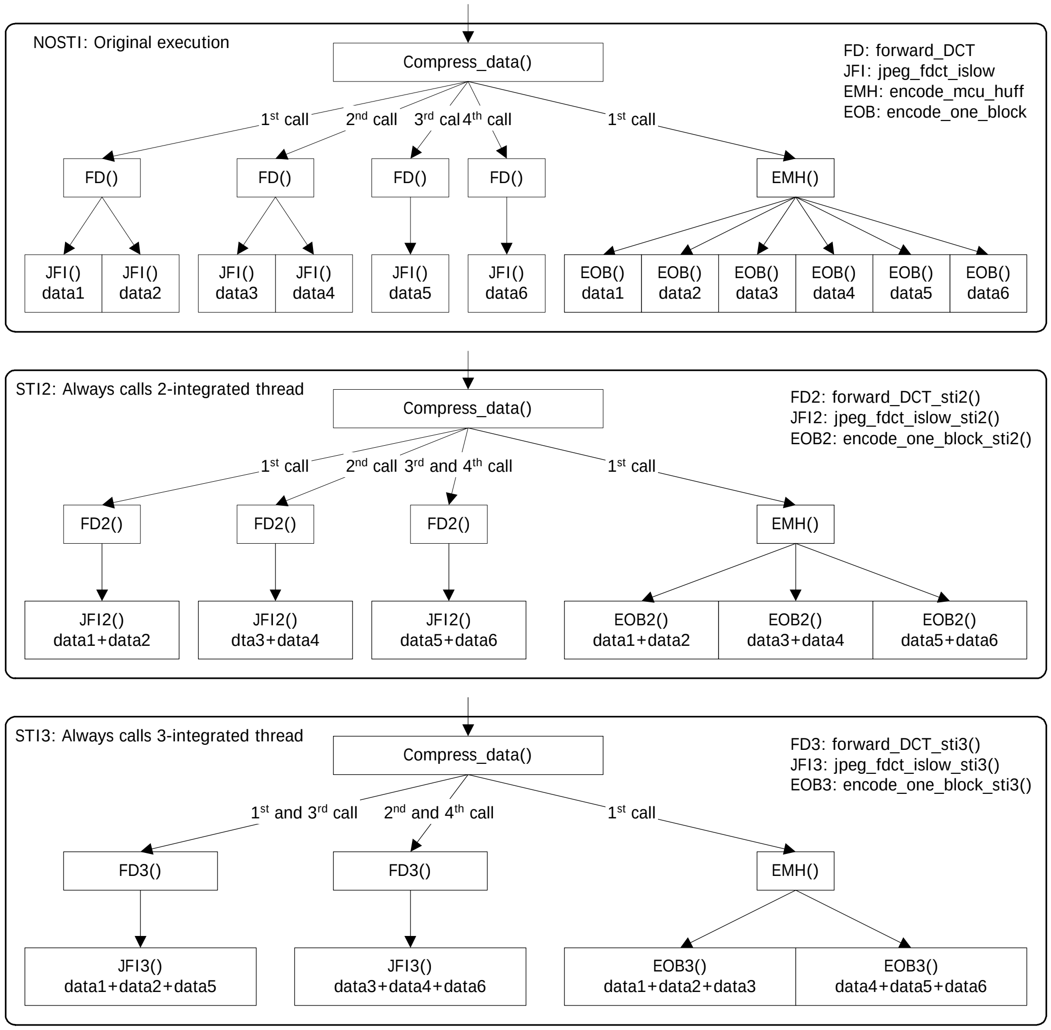


Figure 3.3. 'Direct call' execution models for FDCT and EOB in CJPEG application

3.3. Overview of the experiment and evaluation methods

Three versions of source code (1/ NOSTI: original discrete version, 2/ STI2: 2-thread-integrated, 3/ STI3: 3-thread-integrated) for respective target procedures (IDCT in DJPEG, FDCT and EOB in CJPEG) are written and compiled with various compilers with different optimization options: GCC –O2, Pro64 –O2, ORCC –O2 and –O3, Intel –O2, –O3, and –O2u0. (–O2 without loop unrolling) GCC is the compiler bundled in Linux/IA-64 and Pro64™ is the open source compiler developed by SGI. ORCC is Open

Research Compiler evolved from Pro64 and Intel C++ compiler is a commercial compiler released by Intel Corporation. Table 3.1 lists the compilers which we used in this experiment. The main reason for using various compilers is that the performance of a program varies significantly with the compiler in VLIW/EPIC architectures because scheduling decisions are made at compile time. The second is that we try to observe the correlation between features of the compliers and the performance benefits of STI. Figure 3.4 presents the overview of the experiment.

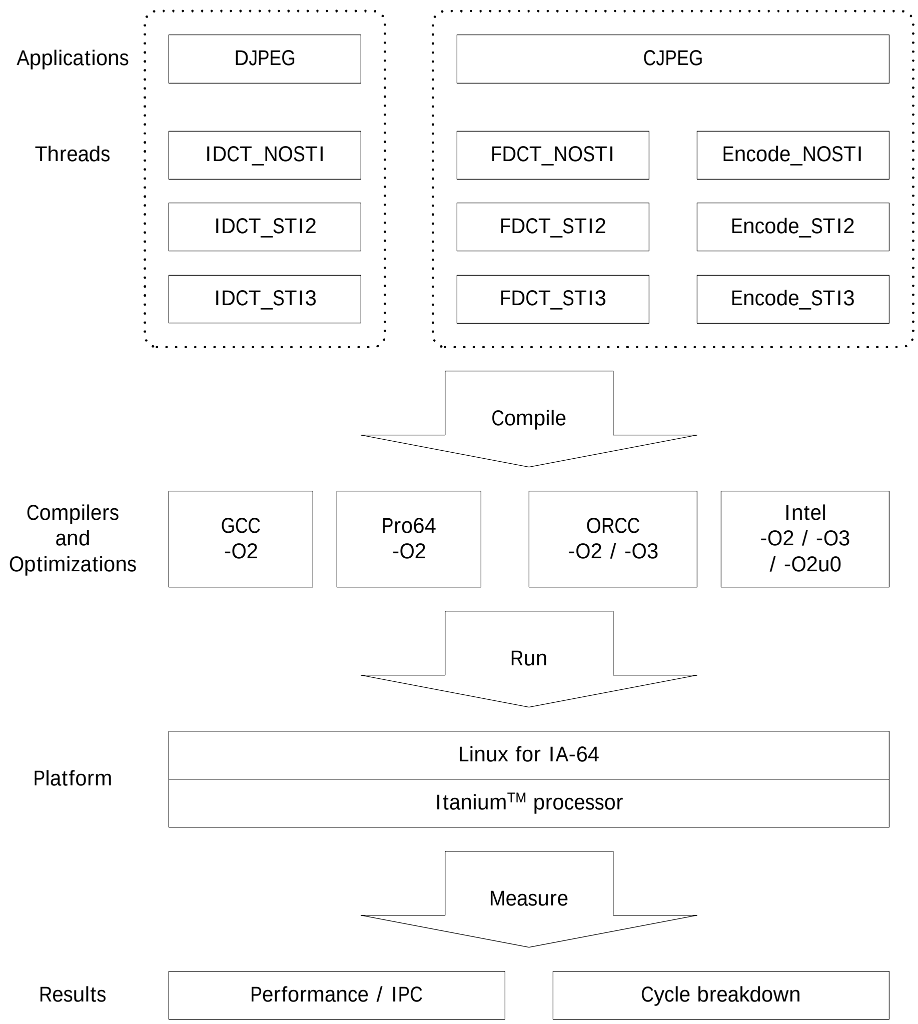


Figure 3.4. Overview of the experiment

Symbol	Name	Version	License
GCC	GNU C Compiler for IA-64	3.1	GNU, Open Source
Pro64	SGI Pro64™ Compiler	Build 0.01.0-13	SGI, Open Source
ORCC	Open Research Compiler	Release 1.0.0	Open Source
Intel	Intel C++ Compiler	6.0	Intel, Commercial

Table 3.1. Compilers used in the experiment

The compiled programs are executed on an Intel Itanium™ processor running Linux for IA-64. Intel Itanium™ processor is the implementation of IA-64 architecture proposed by Hewlett-Packard and Intel. It features EPIC (Explicitly Parallel Instruction Computing), predication, and speculation. It can issue a maximum of 6 instructions per clock cycle and has 2 integer, 2 memory, 3 branch, and 2 floating-point functional units. Six instructions are encoded into a Bundle with stops and a template. A stop splits dependent instructions and a template determines the dispersal from issue slots to functional units. Itanium™ has 3 levels of caches, 16K L1 data cache, 16K L1 instruction cache, 96K unified L2 cache, and 2048K unified L3 cache. It runs with an 800MHz CPU clock rate and 200MHz memory bus speed. [Intel00]

All experimental data are captured during execution the help of the Performance Monitoring Unit (PMU) in Itanium processor. The PMU features hardware counters, which enable the user to monitor a specific set of pipeline events. The software tool and library *pfmon* make it easy to access the PMU simply by adding macros or procedure calls to the original source code. [ME02] We measure the performance (execution cycles or time), instruction per cycle (IPC), and cycle breakdown of the procedures, which shows how execution cycles consist of inherent execution cycles and specific kinds of stalls. All data are obtained by averaging results from 5 execution runs; there is a little variation among the data.

4. Experimental Results

Three kinds of data were obtained to observe the performance and execution behavior of the integrated threads.

(1) CPU cycles, speedup by STI, and IPC

CPU cycles (or execution time) of different versions of the respective target procedures are measured and normalized compared with the performance of the original function compiled with GCC-O2 so that it indicates performance. Percentage speedup is plotted comparing the performance of the integrated version with the original one. We also measure number of instructions retired and compute IPC.

(2) Cycle and speedup breakdown

Every cycle spent on running program on Itanium can be separated in two categories: The first is an 'inherent execution cycle', a cycle used to do the real work of the program and the other is a 'stall', the cycles lost waiting for a hardware resource to become available. The stall can be also subdivided to seven categories: Data access, dependencies, RSE activities, Issue limit, Instruction access, Branch re-steers, and Taken branches. Table 4.1 shows how each category is related to the specific pipeline event. [Intel01]

We measure the cycle breakdown of the each procedure for identifying the benefits and bottlenecks of STI. From those data, we also derive and plot percentage speedup breakdown showing from which categories a performance increase or decrease occurs. By adding numbers of bars with the same color in that chart, we find the overall speedup from STI. The categories which have positive bars contribute to speedup, and those with negative bars cause slowdown.

(3) Code size

STI causes code size increase. We measured the pure code size of the procedure with encoded bundle size excluding data space.

Categories	Descriptions
Inh. Exe. (Inherent execution)	Cycles due to the inherent execution of the program
Inst. Acc. (Instruction access)	Instruction fetch stalls due to L1 I-cache or TLB misses
Data Acc. (Data access)	Cycles lost when instructions stall waiting for their source operands from the memory subsystem, and when memory flushes arise
RSE (RSE activities)	Stalls due to register stack spills to and fills from the backing store in memory
Dep. (Scoreboard dependencies)	Cycles lost when instructions stall waiting for their source operands from non-load instructions
Issue Lim. (Issue limit)	Dispersal break due to stops, port over-subscription or asymmetries
Br. Res. (Branch resteer)	Cycles lost due to branch misperdictions, ALAT flushes, serialization flushes, failed control speculation flushes, MMU-IEU bypasses and other exceptions
Taken Br. (Taken branches)	Bubbles incurred on correct taken branch predictions

Table 4.1. Itanium™ cycle breakdown categories

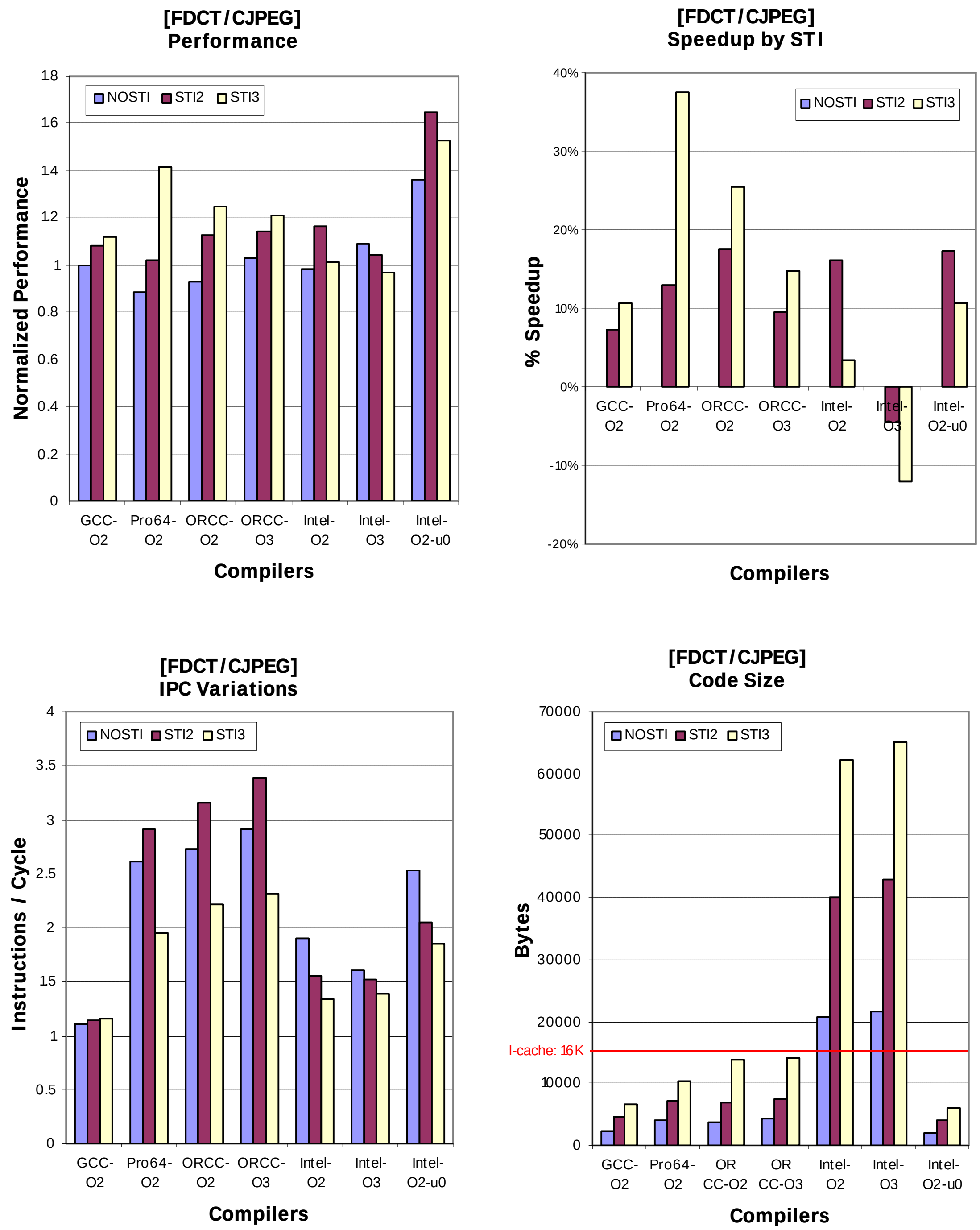


Figure 4.1. Performance, speedup by STI, IPC, and code size of FDCT in CJPEG

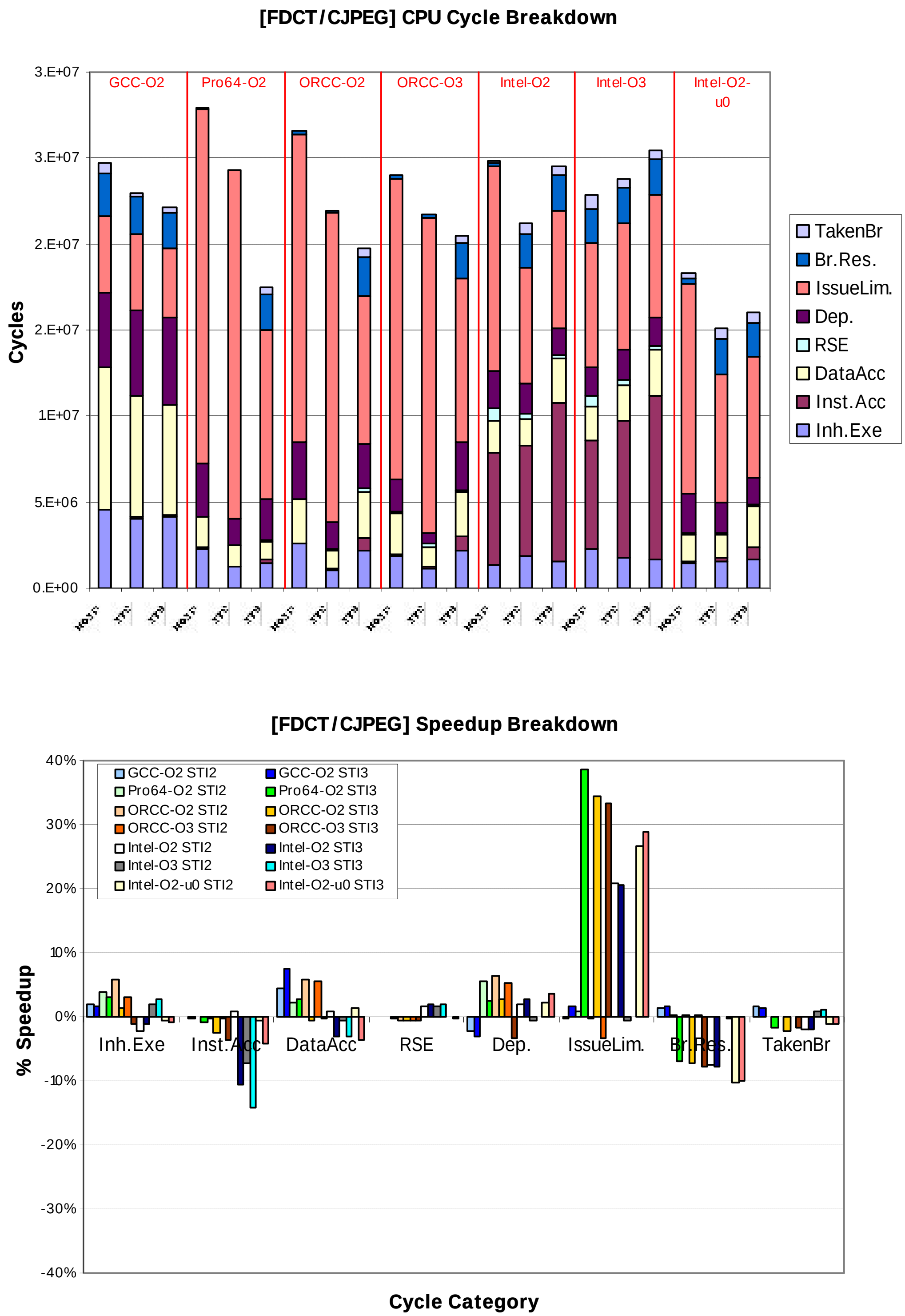


Figure 4.2. Cycle and speedup breakdown of FDCT in CJPEG

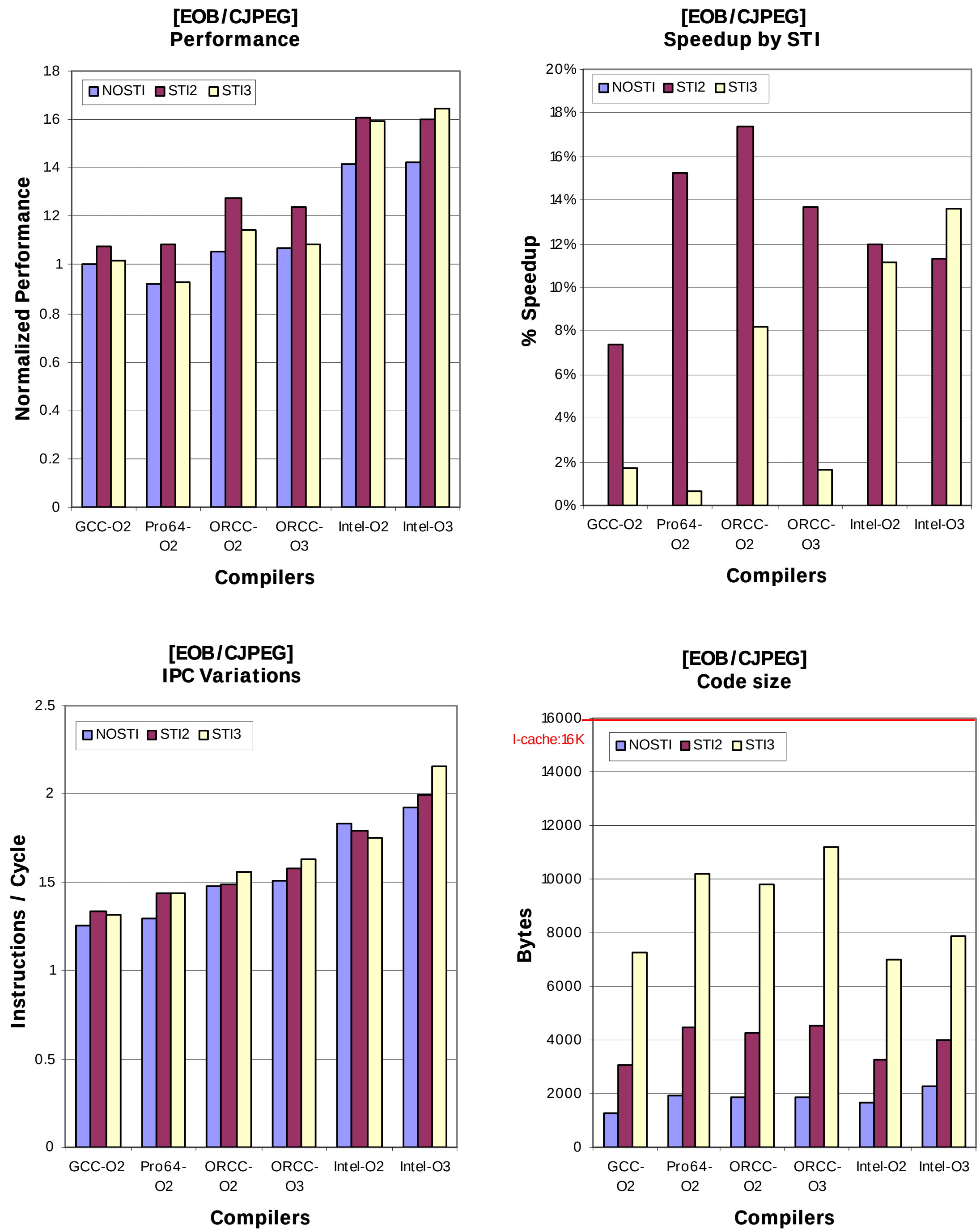


Figure 4.3. Performance, speedup by STI, IPC, and code size of EOB in CJPEG

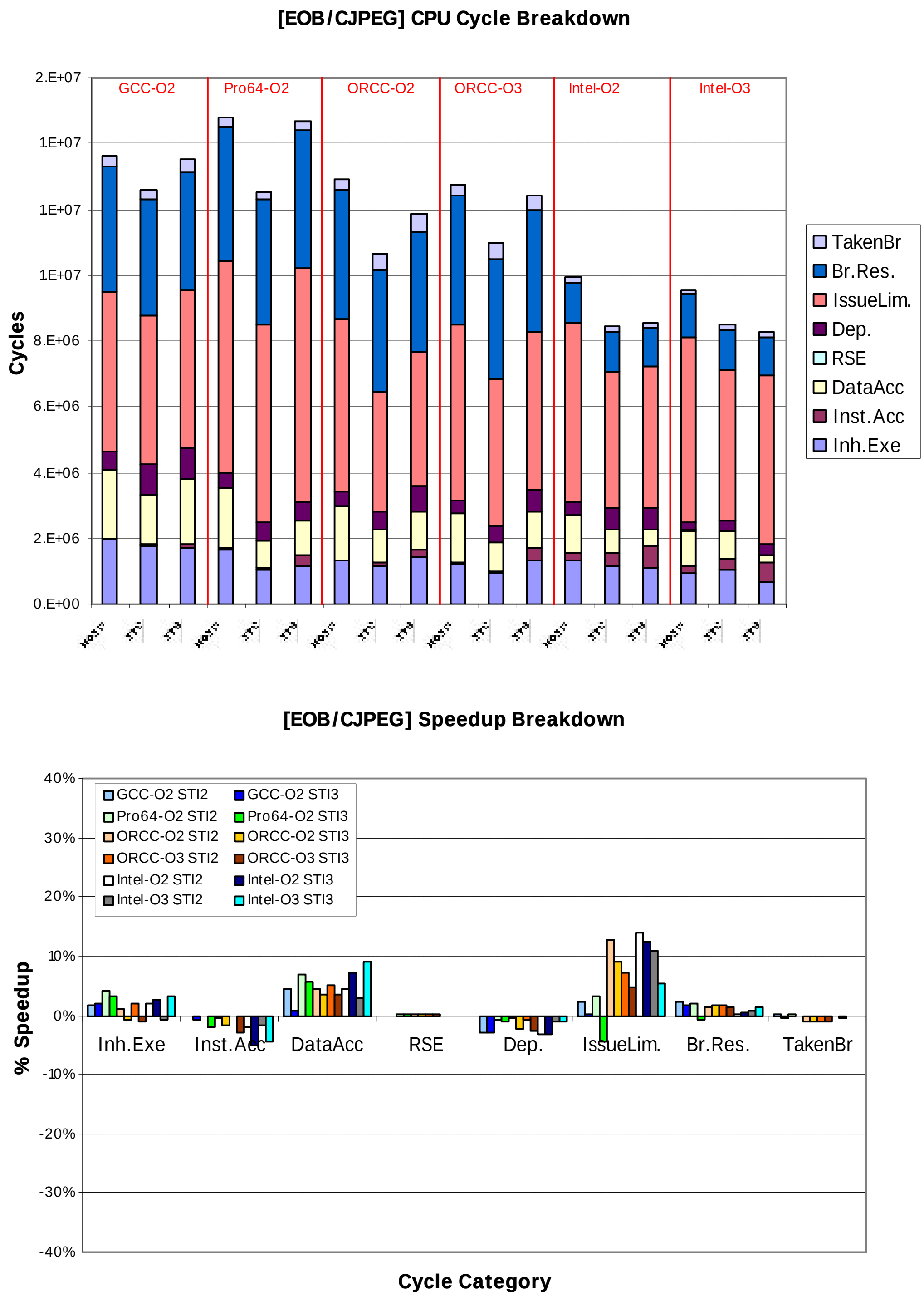


Figure 4.4. Cycle and speedup breakdown of EOB in CJPEG

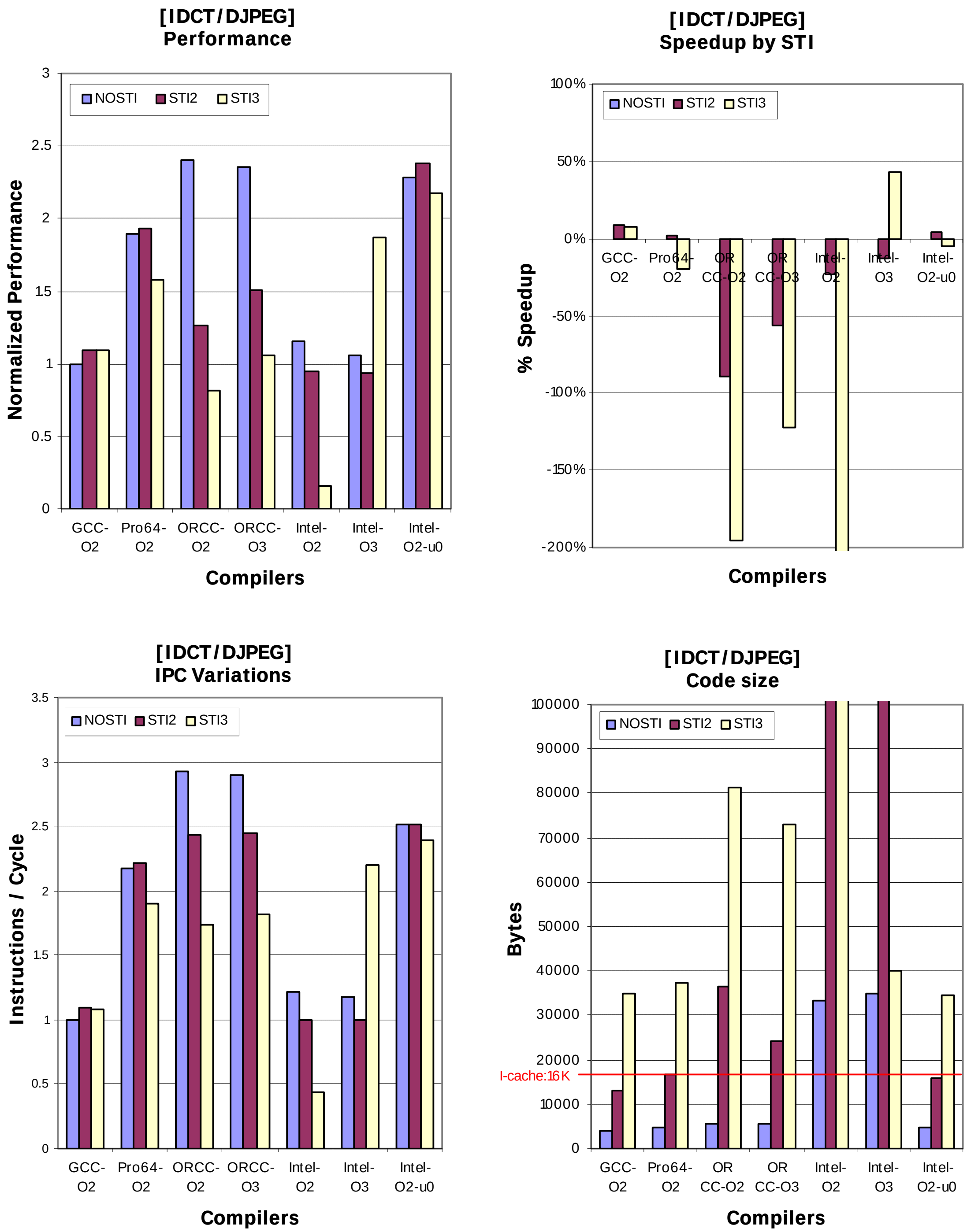


Figure 4.5. Performance, speedup by STI, IPC, and code size of IDCT in DJPEG

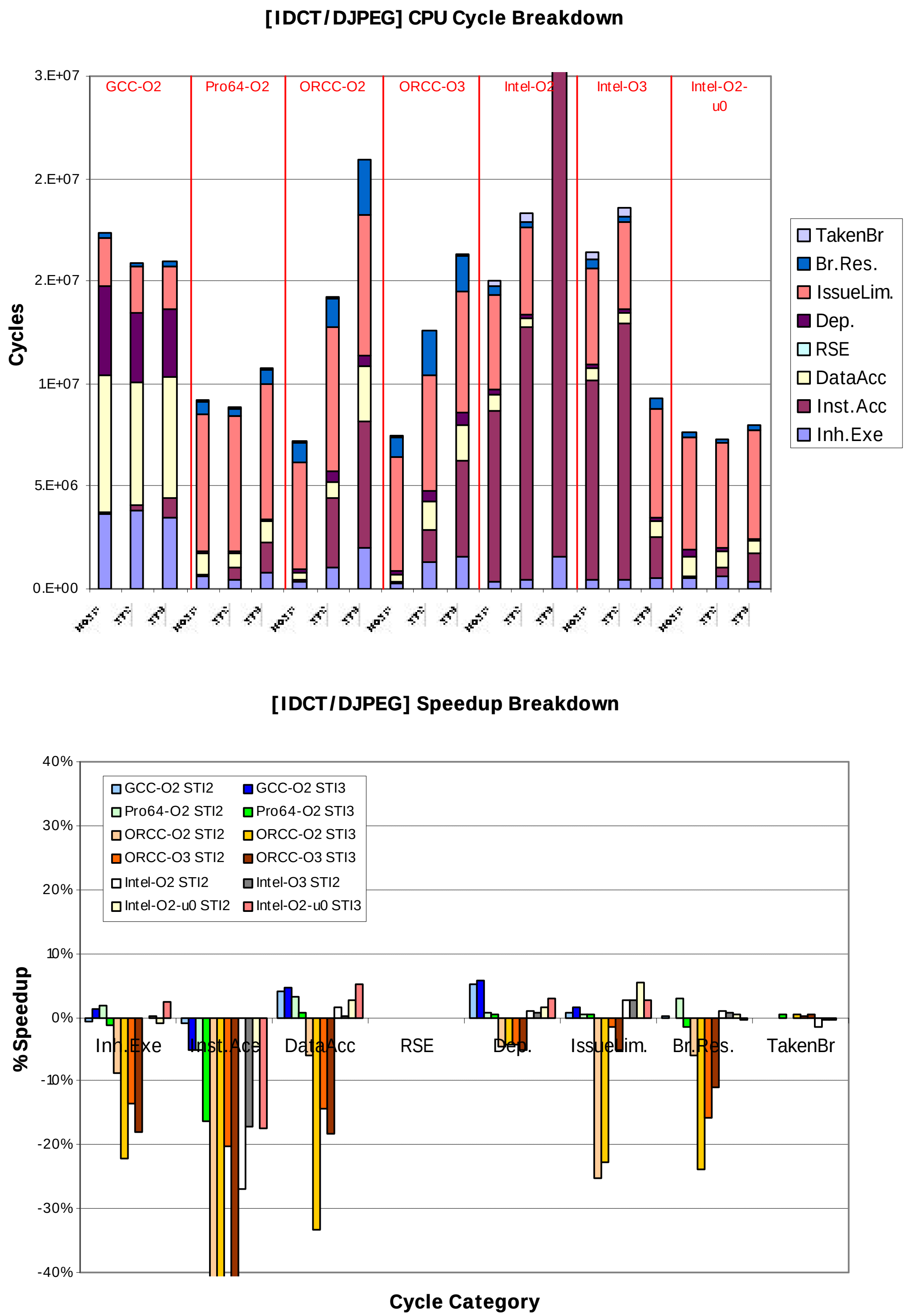


Figure 4.6. Cycle and speedup breakdown of IDCT in DJPEG

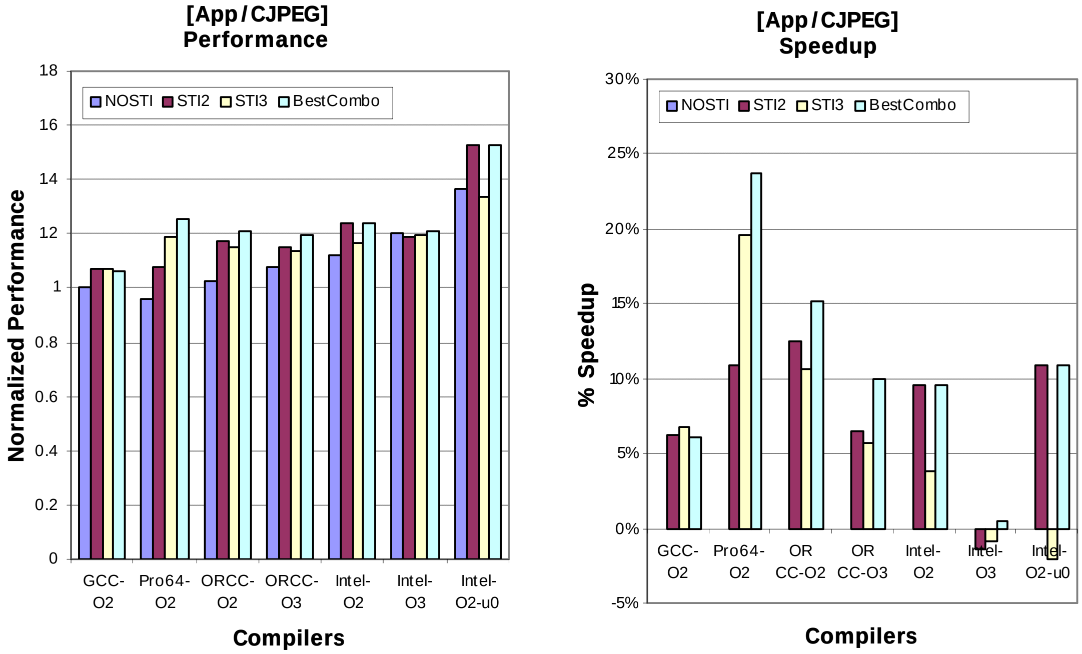


Figure 4.7. Performance and speedup by STI of CJPEG application

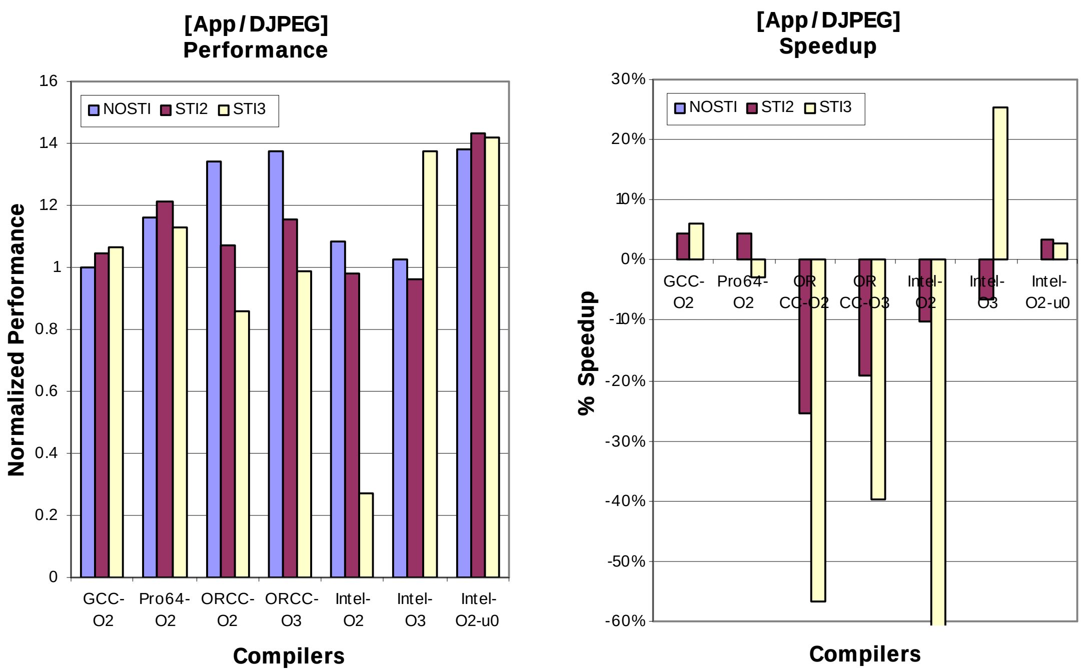


Figure 4.8. Performance and speedup by STI of DJPEG application

4.1. Integrated thread 1: FDCT in CJPEG

Figure 4.1 gives the performance, speedup by STI, IPC variations and code size of 3 versions (NOSTI, STI2, STI3) of FDCT procedure in CJPEG application. Performance varies by compilers because they generate different instruction streams for respective optimization strategies. The original procedure shows the best performance with the compiler Intel-O2-u0. In all cases but Intel-O3, the performance of integrated versions exceed that of the original versions; By integrating 2 threads, it achieves from –4.56% to +17.37% speedup and from –11.97% to +37.49% by integrating 3 threads. 2-integrated version (STI2) compiled with Intel-O2-u0 shows the best performance from all variations, achieving 17.37% speedup from the original version (NOSTI).

STI increases code size; here we see code size increase between 74.63% and 100.68% by STI2 and between 162.90% and 254.55% by STI3. Intel-O2 and Intel-O3 perform loop unrolling and generated codes by those result in the code size much bigger than L1 I-cache size. Intel-O2-u0, which performs the same optimization as -O2 but disabling loop unrolling, generate smaller code and achieves better performance than other compilers. Comparing the speedup and code size, we can see code size is closely related with the performance. When the size of the code is much bigger than the L1 I-cache (Intel-O2 and Intel-O3), performance is degraded after integration. IPC is not closely correlated with the performance because number of instructions generated by compilers varies much by compilers. IPC cannot be accepted as a measure of processor utilization in this case.

Figure 4.2 shows the CPU cycle breakdown and speedup breakdown of FDCT in CJPEG application. The composition of categories is different by compilers because each compiler has its own optimization strategy. However, observing the changes occurring after integration gives basic information about why and how integration achieves speedup. The second bar chart in Figure 4.2 shows the sources of speedup (bars above the centerline) and slowdown (below it). Most of speedup results from reducing issue-limited cycles, showing how the compilers are able to generate better schedules when given more independent instructions. Some improvement comes from reducing data cache misses as well. Major sources of slowdown are stalls from instruction access and branches (either taken or not). This is reasonable because integration increases code size and number of branches. Stalling on instruction access is the most significant factor in Intel-O2 and Intel-O3 because the code size is much bigger than the I-cache size. In such cases, integration results in performance degradation.

4.2. Integrated thread 2: EOB in CJPEG

Figure 4.3 gives the performance, speedup by STI, IPC variations and code size of 3 versions (NOSTI, STI2, STI3) of EOB procedure in CJPEG application. The original procedure shows the best performance with Intel-O3. Performance of the integrated versions always exceeds that of the original ones and STI2 is better than STI3 but Intel-O3 case; It achieves 7.38% ~ 17.38% speedup by STI2, 0.69% ~ 13.61% by STI3. 3-integrated version (STI2) of EOB compiled with Intel-O3 shows the best performance from all variations, achieving 13.61% speedup from the original version (NOSTI).

Code size excessively increases after the integration; it varies between 80.00% ~140.68% by STI2 and 251.43% ~ 491.53% by STI3. However, it is always smaller than I-cache size, which is different from FDCT cases. IPC tends to increase but is not strongly correlated with the performance.

Figure 4.4 shows the CPU cycle breakdown and speedup cycle breakdown of EOB in CJPEG application. The main source of speedup is reducing stalls from issue limit and data access. Stalls from instruction access, dependencies and taken branch tend to increase after integration. However, stalls from instruction access remains small portion of the total execution cycles compared with those of FDCT because the code size always remains smaller than L1 I-cache size.

4.3. Integrated thread 3: IDCT in DJPEG

Figure 4.5 gives the performance, speedup by STI, IPC variations and code size of 3 versions (NOSTI, STI2, STI3) of IDCT procedure in DJPEG application. Unlike the FDCT and EOB cases, integrated versions show speedup with GCC-O2, Pro64, Intel-O3 and Intel-O2-u0 but do not with others; Speedup varies between -89.85% and 8.35% by STI2 and between -617.89% and 43.40% by STI3. STI3 by Intel-O3, which shows the highest performance improvement, is considered an anomaly because the compiler might fail to apply all possible optimizations due to too big code size. Though ORCC does the best job compiling the original version, it does a poor job compiling the integrated versions. The performance of the integrated versions compiled by Intel compiler also suffers except when loop unrolling is disabled. Intel-O2-u0 achieves 3.94% speedup by STI2.

Due to the large code size of the original procedure and the large data-dependent code block in IDCT, code size increase is significantly higher than the previous cases; It is 222.37% ~ 554.60% by STI2 and 14.13% ~ 1644.24% by STI3, which leads to the size much bigger than I-cache size. Every single case but Intel-O3 where the code size exceeds L1 I-cache size results in performance degradation. It indicates that STI is not always useful for improving the performance. If the code size of the original procedure is large and code size expansion rate is high, it limits the general use of integration. However, the Intel-O2-u0 case, where the original version shows the best performance except ORCC cases, shows 3.94% speedup by integrating 2 threads. This shows that we can take advantage of integration by controlling optimizations in some cases. Disabling loop unrolling result in significant performance improvement of original and integrated versions in this case by reducing the instruction cache misses.

The cycle breakdown in Figure 4.6 shows that the portion of the stalls from instruction access is a significant factor in the cases where the code size is much bigger than L1 I-cache size. The speedup breakdown presents that integrated versions compiled by ORCC increases most kinds of stalls. Other than ORCC cases, integration tends to decrease inherent execution cycles and stalls from data access, dependencies, and issue limits. Though there are some sources for speedup, slowdown by increase of instruction cache misses is too crucial so that it fails to achieve speedup by STI in most cases.

4.4. Overall Performance Benefit

These results show how integration helps the performance of parallel threads in JPEG application. Threads FDCT and EOB in CJPEG generally take advantage of integration though the thread IDCT in

DJPEG does not. Since integrated versions of two procedures in CJPEG show a significant speedup, we can optimize CJPEG application by invoking the most efficient thread. NOSTI in Figure 4.7 represents execution of the original version for both EOB and FDCT, and STI2 shows the 2-thread-integrated version for both and vice versa. There are nine combinations of choosing 3 versions of threads for both EOB and FDCT. We also build 'BestCombo' by choosing the best combination for respective compilers based on the performance results of the respective procedures and measure the performance. Table 4.2 shows how the best combinations are chosen for respective compilers.

Figure 4.7 shows performance and speedup by STI of CJPEG application. BestCombo for GCC-O2 has a little noise in data, which is supposed to be the caller side overhead not included in measurement of the performance of the respective procedures. Except that case, performance improvement by choosing the best combination is significant. Maximum speedup of CJPEG by choosing the best combination for respective compilers varies from 0.45% to 23.78%. Intel-O2-u0 shows the best performance for original version and BestCombo, which achieves 10.88% speedup by STI.

Figure 4.8 presents performance and speedup by STI of DJPEG application. Integrated versions compiled by ORCC and Intel shows slowdown because integration fails to improve the performance of IDCT. Intel-O2-u0 case shows the best performance of original version and 2-integrated version achieves the best performance from all variations, improving 3.54% performance by STI.

Compiler	**Best FDCT**	**Best EOB**	**BestCombo**
GCC –O2	STI3	STI2	FDCT_STI3 + EOB_STI2
Pro64 –O2	STI3	STI2	FDCT_STI3 + EOB_STI2
ORCC –O2	STI3	STI2	FDCT_STI3 + EOB_STI2
ORCC –O3	STI3	STI2	FDCT_STI3 + EOB_STI2
Intel –O2	STI2	STI2	FDCT_STI2 + EOB_STI2
Intel –O3	NOSTI	STI3	FDCT_NOSTI + EOB_STI2
Intel –O2u0	STI2	STI2	FDCT_STI2 + EOB_STI2

Table 4.2. Best Combination to invoke threads in CJPEG application

4.5. Compilers and Performance

Compilers other than GCC – 'good' compilers – have many optimization features for IA-64 architecture such as predication and speculation. Due to those optimizations, number of instructions generated by those is bigger than that by GCC. This results in the bigger code size, lager number of instructions retired and the higher IPC except the cases where there is a significant amount of I-cache stalls. For example, the portion of data access stalls of FDCT in CJPEG (in Figure 4.2) significantly decreases with good compilers, which may be the benefit of data speculation or better data layout in memory. On the other hand, more instructions cause the traffic on the issue slots and increase the stalls from issue limit. Absolute performance of both CJPEG and DJPEG is generally better with good compilers and the best with Intel-O2-u0. (See Figure 4.7 and 4.8) It demonstrates that optimizations like loop unrolling should be carefully applied when we performs STI. Whether it is ST2 or STI3, maximum speedup by STI usually higher than that of GCC in most cases. However, performance degradation is also more distinct with good compilers when code size is much bigger than I-cache size.

5. Conclusions and Future Work

This thesis presents the general methodology to use Software Thread Integration (STI) for converting abundant TLP (Thread level parallelism) in a multimedia application to ILP (Instruction level parallelism) for improving utilization of VLIW/EPIC processors. These techniques convert procedure level parallelism to the instruction level, improving performance enabling the better schedule with more independent instructions. It also introduces the ‘Smart RTOS’ concept as a dynamic execution model for exploiting the performance of integrated threads. This work is especially relevant to high-end embedded processors running multimedia applications with ILP support. STI could improve utilization of uniprocessors with no additional hardware by executing multiples threads concurrently.

We demonstrate our technique by integrating three procedures from JPEG application, compiling with four compilers for the Itanium™ EPIC architecture and measuring the performance with the on-chip performance measurement units. When compared with optimized code generated by the best compiler without our methods, we find procedure speedup of up to 18% and program speedup up to 11%. Detailed performance analysis shows the primary bottleneck to be the Itanium’s 16K instruction cache, which has limited room for the code expansion introduced by thread integration.

This research is the first step for the big picture of STI and the ‘Smart RTOS’. We have developed the methods to plan and perform STI focusing on specific level of parallelism – procedure-level data parallelism – abundant in multimedia applications. From the experiment with the sample application, we have demonstrated a significant amount of performance improvement by STI and have identified that the code size is the most crucial factor which limits the performance. Future research will extend the proposed methodology to the different kinds of threads with more various levels of parallelism. It will also examine the benefits and bottlenecks of STI with other realistic multimedia workloads such as multiple different applications or multiple instances of the same application. It will involve developing a tool which automates the integration process at the assembly level like *Thrint*, the post-pass compilation tool already developed. The detailed model and algorithm for a ‘Smart RTOS’ will be built up and investigated.

References

[AD98] H. Akkary and M. A. Driscoll. “A Dynamic Mutithreading Processor,” Proceedings of 31st Annual International Symposium on Microarchitecture, Nov. 1998.

[BCG+95] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. “An Overview of the PARADIGM Compiler for Distributed-Memory Multicomputers,” IEEE Computer, Vol. 28, No. 10, pages 37-47, Oct. 1995.

[CHK93] K. D. Cooper, M.W. Hall, K. Kennedy. A methodology for procedure cloning. Computer Languages, Vol. 19, Issue. 2, pages 105-117, Feb. 1993

[DCG94] J. Dean, C. Chambers and D. Grove. “Selective specialization for object-oriented languages,” ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 93-102, June 1995.

[Dean00] Alexander G. Dean, “Software Thread Integration for Hardware to Software Migration,” Doctoral Thesis, Carnegie Mellon University, Pittsburgh, PA, May 2000.

[Dean02] Alexander G. Dean, “Compiling for Fine-Grain Concurrency: Planning and Performing Software Thread Integration,” 6th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-6, in conjunction with HPCA 8), Feb. 2002.

[DFGS00] Benoit Dupont de Dinechin, Francois de Ferriere, Chrisophe Guillon and Arthur Stoutchinin. “Code Generator Optimizations for the ST100 DSP-MCU Core,” Proceedings of CASES 2000, pages 93-102, Nov. 2000.

[DS98] Alexander G. Dean and John Paul Shen, “Techniques for Software Thread Integration in Real-Time Embedded Systems,” Proceedings of the 24th EUROMICRO Conference, Aug. 1998.

[Ersh66] A.P. Ershov, “ALPHA – An Automatic Programming System of High Efficiency,” Journal of ACM, Vol. 13, Issue 1, pages 17-24, Jan. 1966.

[Fish81] J. Fisher. “Trace scheduling: A technique for global microcode compaction,” IEEE Transactions on Computers C-30, Vol. 7, pages 478-490, July 1981.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization,” ACM Transactions on Programming Languages, Vol. 9, Issue 3, pages 319-349, July 1987.

[FWL99] J. Fritts, W. Wolf and B. Liu. “Understanding multimedia application characteristics for designing programmable media processors,” Proceedings of SPIE, Vol. 3655, pages 2-13, Jan. 1999.

[GKM82] S. L. Graham, P. B. Kessler and M. K. McKusick. “gprof: a call graph execution profiler,” Proceedings of SIGPLAN’82 symposium on Compiler Construction, pages 120-126, June 1982.

[HA99] J. Hoogerbrugge and L. Augusteijn. “Instruction scheduling for TriMedia,” Journal of Instruction-Level Parallelism, Vol. 1, Issue 1, Feb. 1999.

[Half99] T.R. Halfhill, “StarCore Reveals Its First DSP,” Microprocessor Report, Vol. 13, No. 6, pages 13-16, May 1999.

[Hall91] M. W. Hall. Managing Interprocedural Optimization, Ph.D. thesis, Rice University, Apr. 1991

[Hals01] Fred Halsall, *Multimedia communications: Applications, Networks, Protocols and Standards*, Pearson Education Limited, 2001

[HBC98] W. A. Havanki, S. Banerjia, and T. M. Conte, “Treegion Scheduling for Wide Issue Processors,” Proceedings of. 4th International Symposium on High-Performance Computer Architecture, pages 266-276, Feb. 1998.

[Hwu93] W. M. Hwu, et. al. “The superblock: An effective technique for VLIW and superblock compilation,” Journal of Supercomputing, Vol. 7, pages 229--248, Jan. 1993.

[Intel00] Intel IA-64 Architecture SoftwareDeveloper’s Manual, Vols. I-IV, Rev. 1.1, Intel Corp., July 2000; http://developer.intel.com.

[Intel01] Intel IA-64 Architecture SoftwareDeveloper’s Manual Specification Update, Intel Corp., Aug. 2001; http://developer.intel.com.

[Lam98] M. Lam, “Software pipelining : an effective scheduling technique for VLIW machines,” Proceedings of ACM SIGPLAN Conference Programming Language Design and Implement, pages 318-328, June 1988.

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. “Mediabench: a tool for evaluating and synthesizing multimedia and communications systems,” Proceedings of 30th annual ACM/IEEE international symposium on Microarchitecture, Dec. 1997.

[MCHB92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. “Effective compiler support for predicated execution using the hyperblock,” Proceedings of the 25th International Symposium on Microarchitecture, pages 45-54, Dec. 1992.

[ME02] David Mosberger and Stephane Eranian, *IA-64 Linux Kernel: Design and Implementation*, Prentice Hall PTR, 2002.

[MME+97] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen and A. Polyak. “Simulation/evaluation environment for a VLIW processor architecture,” IBM Journal of Research and Development, Vol. 41, No. 3, Mar. 1997.

[MOAV99] I. Martel, D. Ortega, E. Ayguade, and M. Valero, “Increasing Effective IPC by exploiting Distant Parallelism,” Proceedings of International Conference on Supercomputing, pages 348-355, June 1999.

[Newb97] Chris J. Newburn, “Exploiting Multi-Granied Parallelism for Mulitple-Instruction Stream Architectures,” Doctoral Thesis, Carnegie Mellon University, Pittsburgh, PA, Nov. 1997.

[OCS01] E. Özer , T. M. Conte and S. Sharma. “Weld: A Multithreading Technique towards Latency-tolerant VLIW Processors,” Proceedings of 8th International Conference on High Performance Computing, Dec. 2001.

[Purc97] S. Purcell, “Mpact2 media processor, balanced 2X performance,” Proceedings of the SPIE Symposium on Multimedia Hardware Architecture, pages 102-108, Feb. 1997.

[PV00] Gerald G. Pechanek and Stamatis Vassiliadis. “The ManArray Embedded Processor Architecture,” Proceedings of the 26th Euromicro Conference: “Informatics: inventing the future”, Vol. I, pages 348-355, Sep. 2000.

[Ryde82] Babara G. Ryder. “Incremental data flow analysis based on a unified model of elimination algorithms,” Ph.D. thesis, Dept. of Computer Science, Rutgers Univ., New Brunswick, NJ, 1982.

[SBV95]G. S. Sohi, S. Breach, and T. N. Vijaykumar, “Multiscalar Processors,” Proceedings of 22nd International Symposium of Computer Architecture, pp. 414--425, June 1995.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. “Simultaneous multithreading: Maximizing on-chip parallelism,” Proceedings of 22nd International Symposium on Computer Architecture, May 1995.

[TH97] J. Turley and H. Hakkarainen. “TI’s New ‘C6x DSP Screams at 1,600 MIPS,” The Microprocessor Report, Vol. 11, No. 2, pages 14-17, Feb. 1997.

[THA+99] Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P. C. Yew. “The Superthreaded Processor Architecture,” IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems, pages 881-902, Sep. 1999.

[Wall91] David W. Wall. “Limits on instruction-level parallelism,” Proceedings of 4th International Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS4), pages 176-188, April 1991.

[Wall92] Gregory K. Wallace, “The JPEG Still Picture Compression standard,” IEEE Transactions on Consumer Electronics, Vol. 38, Issue 1, Feb. 1992.

[WS90] S. Weiss and J. E. Smith. “A Study of Scalar Compilation Techniques for Pipelined Supercomputers,” ACM Transactions On Mathematical Software, Vol. 16, Issue 3, pages 223-245, Sep. 1990

[ZVSM94] V. Zivojnovi'c, J. Mart ' inez, C. Schlager, and H. Meyr, “DSPstone: A DSP-oriented benchmarking methodology,” Proceedings of ICSPAT’94, Oct. 1994.

Appendix A. Data Tables

Experimental results are based on the values in the following tables.

		GCC-O2	Pro64-O2	ORCC-O2	ORCC-O3	Intel-O2	Intel-O3	Intel-O2-u0
NOSTI	Cycles	24745201.4	27936141.2	26592293.8	24038848.2	25258670.6	22743483.4	18129964.2
	Inst	27369441	72831033.6	72417389.2	70020903.6	48152661.6	36635502.2	45838180.4
	CodeSize	2368	3968	3872	4288	20768	21552	2080
	IPC	1.106050444	2.607054177	2.723247184	2.912822737	1.90638147	1.610813153	2.528310585
	Abs.Perf	1	0.88577736	0.930540313	1.02938382	0.979671567	1.088012815	1.364878669
	Impr.STI	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
STI2	Cycles	22952715	24323655	21943601.2	21727335.2	21189165	23779515	14980846.6
	Inst	26346684.6	70720522.6	69208824.8	73644869.2	32767701.8	36142999	30643674
	CodeSize	4752	7168	7040	7488	40048	42896	4000
	IPC	1.147867893	2.907479267	3.153941059	3.389503062	1.546436672	1.519921622	2.045523515
	Abs.Perf	1.078094744	1.017330718	1.127672763	1.138897208	1.167823338	1.040610012	1.651789252
	Impr.STI	7.2438%	12.9312%	17.4814%	9.6157%	16.1113%	-4.5553%	17.3697%
	CS incr.	100.68%	80.65%	81.82%	74.63%	92.84%	99.03%	92.31%
STI3	Cycles	22127275.2	17463514.4	19826513.2	20492482.6	24408994.4	25465858.4	16207522
	Inst	25723121.6	33921646.2	43993159	47353716	32754490.8	35483411.4	29907826.4
	CodeSize	6576	10432	13728	14048	62128	65184	6064
	IPC	1.16250742	1.942429538	2.218905491	2.310784736	1.341902508	1.393371896	1.845305309
	Abs.Perf	1.118312182	1.416965728	1.248086396	1.207525798	1.013773898	0.971701052	1.526772655
	Impr.STI	10.5795%	37.4877%	25.4426%	14.7526%	3.3639%	-11.9699%	10.6037%
	CS incr.	177.70%	162.90%	254.55%	227.61%	199.15%	202.45%	191.54%

Table A.1. Performance data from FDCT in CJPEG

		Inh.Exe	Inst.Acc	DataAcc	RSE	Dep.	IssueLim.	Br.Res.	TakenBr
GCC-O2	NOSTI	4560708.4	31087.2	8226111.2	0	4324572.8	4426993.2	2483811	688187.4
	STI2	4062928.4	36111.6	7117946.6	0	4888797	4499357	2111263	258139.2
	STI3	4178095.8	71201.4	6384021	18596.6	5058683.2	4047999	2047059.2	318985
Pro64-O2	NOSTI	2303495.8	39059	1829287.6	64	3109761	20511182.2	117866.6	6690.8
	STI2	1204700.4	75284	1193108.6	15550.2	1544681	20250046.2	69191.4	4606.2
	STI3	1455578.8	233903.8	1028506	51076.4	2424975.6	9768552.2	2082032.8	438514.6
ORCC-O2	NOSTI	2585197.4	33973.8	2552130.6	28819.8	3281762.6	17847545.4	234497.2	16774
	STI2	1066508.6	76312.8	1012271.6	144386.6	1553236.8	17938220.8	135152.4	6894.2
	STI3	2223411.8	681116	2677555	184288	2565302.4	8684975.8	2187111.2	593287.8
ORCC-O3	NOSTI	1911258	37077.8	2414983.6	37020.8	1892297.6	17537756	209497.4	9661.8
	STI2	1166845	84066.6	1109472	199695.2	653887.4	18330783.2	166398.8	6850.8
	STI3	2172359.6	878258.2	2499180.4	185716.4	2711134.4	9511948.6	2111879.4	392214.8
Intel-O2	NOSTI	1293756.8	6587508.8	1812904.6	715730.8	2228703.8	11924225	165471.2	51488.2
	STI2	1875261	6366613.8	1580733	300603.6	1743870.4	6740772	2024334.2	536606
	STI3	1545048	9192015.8	2584172.4	212334.2	1534354.8	6850094.2	2096230.6	533489.8
Intel-O3	NOSTI	2283636.4	6277913.6	1950588.4	684331.8	1653419.4	7169542.6	2042054.4	777845.4
	STI2	1808248.6	7925781.8	2066976.2	296412	1798826	7319141	2044210.4	572482.6
	STI3	1645807.2	9486637.6	2678532.2	208903	1670223	7167325.8	2099711.8	519739.4
Intel-O2-u0	NOSTI	1484906.4	34415.8	1630733	99683.4	2184078	12303163.8	215333.2	349403.6
	STI2	1585271.2	144626.4	1396208.4	116007.8	1759008	7435311	2076898.8	546301.8
	STI3	1629191.2	777414	2302702.6	172131.4	1528607.4	7008633	2027927.4	536451

Table A.2. Breakdown data from FDCT in CJPEG

		GCC-O2	Pro64-O2	ORCC-O2	ORCC-O3	Intel-O2	Intel-O3
NOSTI	Cycles	13596377.6	14763212.6	12920713.2	12723303.6	9589152.4	9563634.6
	Inst	17013799	19149591.8	19093284.8	19151002.4	17597154.4	18345925
	CodeSize	1280	1920	1856	1888	1664	2240
	IPC	1.251347933	1.297115494	1.477726849	1.505191026	1.835110515	1.918300496
	Abs.Perf	1	0.920963341	1.05229312	1.068620071	1.417891492	1.421674726
	Impr.STI	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
STI2	Cycles	12591708	12508204	10675324.4	10976943.4	8442310	8479175.2
	Inst	16876797.2	18001335.8	15917483.2	17281108.8	15162534.4	16931686.4
	CodeSize	3072	4480	4288	4544	3280	4032
	IPC	1.340310401	1.439162313	1.491053818	1.57430973	1.796017251	1.996855355
	Abs.Perf	1.079788191	1.08699679	1.273626645	1.238630564	1.610504424	1.603502378
	Impr.STI	7.3892%	15.2745%	17.3782%	13.7257%	11.9598%	11.3394%
	CS incr.	140.00%	133.33%	131.03%	140.68%	97.12%	80.00%
STI3	Cycles	13364620	14661989.6	11857405	12518613.2	8520667.2	8262041.2
	Inst	17521906.2	20999315.4	18496309	20413373.4	14953652.2	17796488.8
	CodeSize	7280	10176	9792	11168	6976	7872
	IPC	1.311066547	1.432228229	1.559895188	1.630641755	1.754986065	2.154006301
	Abs.Perf	1.017341129	0.92732146	1.146657097	1.086092955	1.595694009	1.645643888
	Impr.STI	1.7046%	0.6856%	8.2295%	1.6088%	11.1426%	13.6098%
	CS incr.	468.75%	430.00%	427.59%	491.53%	319.23%	251.43%

Table A.3. Performance data from EOB in CJPEG

		Inh.Exe	Inst.Acc	DataAcc	RSE	Dep.	IssueLim.	Br.Res.	TakenBr
GCC-O2	NOSTI	1976498.8	32109.6	2069405.4	403.8	559466	4831644.4	3814154.8	320507.4
	STI2	1766225.8	54550	1464590.6	435.8	966104.2	4510420	3516791.4	315606.6
	STI3	1700684.8	126248.8	1965406	533.4	968446.4	4798243.4	3574775.2	368791.6
Pro64-O2	NOSTI	1656487.4	26437	1866236.2	1493.8	398446.4	6487603	4081152	245193.4
	STI2	1034684.2	62601.6	851609	468.4	506398.4	6030668.2	3804621.6	219561.2
	STI3	1183984.6	310630.4	1049186.2	382.6	561698.8	7125902	4184679.4	279394.8
ORCC-O2	NOSTI	1308909.2	22067.2	1623401.4	296	470265	5239892	3893766.8	359901.4
	STI2	1166175.6	75532	1031624.2	285	547503	3609979.2	3705602.8	499451.6
	STI3	1409887.4	257229.2	1153439	288.4	752649	4082213.2	3666814.2	514512.6
ORCC-O3	NOSTI	1203110.8	38680.4	1530117.4	335.8	359294.6	5381786.4	3886845	326229.4
	STI2	948889.2	66551.6	868669	285.8	470495.4	4467727.4	3687726	447609.8
	STI3	1337826.8	390550.2	1067970.6	149.8	679532.6	4775131.2	3714046.6	453854.8
Intel-O2	NOSTI	1340935.8	178799.8	1195760.2	282.8	351617.8	5506286.4	1208311.8	142544.8
	STI2	1156790.8	379313.4	751143	329.4	661187	4121711.6	1182745	165792.8
	STI3	1097499.2	672460	484752.6	11707.8	659105.2	4282065.2	1155357.4	177698.8
Intel-O3	NOSTI	954058.6	204337.6	1072557.4	6557.8	243072.8	5636812.6	1297036.2	119513.6
	STI2	1034727.6	360584.6	795164.2	12632.6	337122.4	4594474	1218166.8	127998
	STI3	650167.2	621204.2	220798.2	9609.4	331583.8	5136576.4	1151877.8	137123.2

Table A.4. Breakdown data from EOB in CJPEG

		GCC-O2	Pro64-O2	ORCC-O2	ORCC-O3	Intel-O2	Intel-O3	Intel-O2-u0
NOSTI	Cycles	17361704	9176759.4	7213900.6	7378070.8	15023933.8	16388794.4	7601887
	Inst	17434452.8	19996006	21100037	21432325.8	18358867.6	19361617.2	19119191
	CodeSize	3840	4736	5568	5472	33456	34992	4864
	IPC	1.004190188	2.178983357	2.924913742	2.904868546	1.22197474	1.181393623	2.515058564
	Abs.Perf	1	1.89192102	2.406701307	2.353149552	1.155603068	1.059364318	2.283867677
	Impr.STI	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
STI2	Cycles	15912840.6	9000273	13695602.4	11521879.6	18443306	18526065	7302353.2
	Inst	17315283.4	19964241.4	33278657.6	28172230.4	18347011.6	18537322	18388490.2
	CodeSize	13024	16832	36448	24384	118912	119520	15680
	IPC	1.088132775	2.21818176	2.429879068	2.445107168	0.994778897	1.00060763	2.518159516
	Abs.Perf	1.091049954	1.929019709	1.267684582	1.506846504	0.941355308	0.937150118	2.377549199
	Impr.STI	8.3452%	1.9232%	-89.8502%	-56.1639%	-22.7595%	-13.0410%	3.9403%
	CS incr.	239.17%	255.41%	554.60%	345.61%	255.43%	241.56%	222.37%
STI3	Cycles	15960782.6	10982507.8	21322697	16380666.6	107855107.6	9276416.2	7999137.6
	Inst	17285286.2	20816738.2	37151559.6	29871791	47874258	20440255.8	19171136
	CodeSize	34978	37280	81216	73152	583552	39936	34496
	IPC	1.082984878	1.895444882	1.742348053	1.823600451	0.443875669	2.203464717	2.396650359
	Abs.Perf	1.087772726	1.580850596	0.814235835	1.059889956	0.160972479	1.871596059	2.170446974
	Impr.STI	8.0690%	-19.6774%	-195.5779%	-122.0183%	-617.8886%	43.3978%	-5.2257%
	CS incr.	810.89%	687.16%	1358.62%	1236.84%	1644.24%	14.13%	609.21%

Table A.5. Performance data from IDCT in DJPEG

		Inh.Exe	Inst.Acc	DataAcc	RSE	Dep.	IssueLim.	Br.Res.	TakenBr
GCC-O2	NOSTI	3687521.8	43504	6714068	0	4326349.6	2338937.2	232065.6	768.8
	STI2	3819225.8	236180.4	5999889.6	0	3422547.2	2220632.4	194725.2	6204.4
	STI3	3466649.4	925587.2	5926099.2	0	3326653.2	2082888	269684.8	8451.4
Pro64-O2	NOSTI	641440.8	55614	1045728.8	0	121928	6634925	608476.6	51128.6
	STI2	477366.8	539918	746540.6	0	62405	6597015.2	339279.8	55334
	STI3	746418.4	1537928.6	988508.4	0	88065.6	6597469.4	755011.6	14840.8
ORCC-O2	NOSTI	368089.6	89226.8	308430	0	173711.6	5235069.8	976391.8	59825.4
	STI2	1001578	3462091.2	739964.2	0	511547.8	7059369.8	1417636.4	60198.8
	STI3	1970705.6	6168327.4	2717499	0	477857.8	6869465.4	2697532	19956
ORCC-O3	NOSTI	266798.6	64324.2	327045.6	0	206512.2	5528180.4	975332	57627.4
	STI2	1279486	1560301.8	1389048	0	529007	5651485.8	2150924.2	39299.6
	STI3	1594482.2	4681622	1687176.2	0	592998.4	5919846.2	1791503.8	17343.6
Intel-O2	NOSTI	388872.8	8318038.4	723822.8	0	276978.8	4633676.2	427570.6	241407.4
	STI2	395933.4	12352600.2	484505.6	2895.4	126292	4231496.4	275535	457598.6
	STI3	1562471.8	89938934.4	2187457.6	0	747446.6	13837027.4	763284.8	31709.2
Intel-O3	NOSTI	431695.8	9748673.6	551405	0	238214.2	4694067.8	383834.8	349067
	STI2	398997.6	12564584.8	515972	56	121898.8	4271230	256661	438721.2
	STI3	511304.6	1973269.4	855646.6	0	89163.2	5380857.4	485859	15400.2
Intel-O2-u0	NOSTI	537589.6	57609.4	989743	0	293470.6	5506016.4	231109.4	9238.6
	STI2	606560	409286.4	786392.6	2992.4	186097	5100629.6	197388	36020.4
	STI3	367433.2	1384303	588945.8	0	65136.6	5313702.4	260298.2	10999.8

Table A.6. Breakdown data from IDCT in DJPEG

		GCC-O2	Pro64-O2	ORCC-O2	ORCC-O3	Intel-O2	Intel-O3	Intel-O2-u0
NOSTI	Cycles	51111558.8	53400470.6	49749086.4	47602883.8	45597381.2	42424974	37521123.8
	Inst	64073917	118806533	118459363.8	114958043.8	84700261.6	75999128	82396084.4
	CodeSize	223461	283828	293348	306732	481296	484576	465701
	IPC	1.253609135	2.224821835	2.381136466	2.414938647	1.857568557	1.791377126	2.195991912
	Abs.Perf	1	0.957136861	1.027386883	1.073707194	1.120931892	1.204751682	1.362207568
	Impr.STI	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
STI2	Cycles	47941949.6	47596764.2	43554783	44484767.2	41223417.6	43027107	33438963.8
	Inst	62988265	115520558	111950646.2	116673747.4	66920401	74174904	64772819
	CodeSize	228084	291155	300435	314075	502605	508077	472386
	IPC	1.313844462	2.427067469	2.570341039	2.622779768	1.623358879	1.723911022	1.937046237
	Abs.Perf	1.066113482	1.073845243	1.173500481	1.148967658	1.23986709	1.187892061	1.528503069
	Impr.STI	6.2014%	10.8683%	12.4511%	6.5503%	9.5926%	-1.4193%	10.8796%
	CS incr.	2.07%	2.58%	2.42%	2.39%	4.43%	4.85%	1.44%
STI3	Cycles	47642300.6	42940884.2	44451887.6	44872165.4	43879801.2	42780409.4	38273580.8
	Inst	62956531	80528075	89257467	92702483	66641077	74175164	63796367
	CodeSize	234548	301155	313659	327915	528781	534477	481122
	IPC	1.321441874	1.8753241	2.007956733	2.06592399	1.518718754	1.733858208	1.666851276
	Abs.Perf	1.072818864	1.190277279	1.149817512	1.13904819	1.164808349	1.194742162	1.335426624
	Impr.STI	6.7876%	19.5871%	10.6478%	5.7365%	3.7668%	-0.8378%	-2.0054%
	CS incr.	4.96%	6.10%	6.92%	6.91%	9.87%	10.30%	3.31%
BestCombo	FDCT	STI3	STI3	STI3	STI3	STI2	NOSTI	STI2
	EOB	STI2	STI2	STI2	STI2	STI2	STI3	STI2
	Cycles	47967477.2	40703174.8	42196876.4	42856187	41223417.6	42233090.6	33438963.8
	Inst	62300061	77854271	86642618.6	89532642	66920401	75350952	64772819
	CodeSize	230004	294467	307171	320531	502605	490575	481122
	IPC	1.298797949	1.912732149	2.053294604	2.089141575	1.623358879	1.78416855	1.937046237
	Abs.Perf	1.065546111	1.255714304	1.211264036	1.192629638	1.23986709	1.210225396	1.528503069
	Impr.STI	6.1514%	23.7775%	15.1806%	9.9714%	9.5926%	0.4523%	10.8796%
	CS incr.	2.93%	3.75%	4.71%	4.50%	4.43%	1.24%	3.31%

Table A.7. Performance data from CJPEG application

		GCC-O2	Pro64-O2	ORCC-O2	ORCC-O3	Intel-O2	Intel-O3	Intel-O2-u0
NOSTI	Cycles	34328864.8	29603620.8	25638319.8	24936869	31725190.8	33472172.8	24875635.2
	Inst	44749076	55011888	55774157.8	55647595.6	42908257.6	46874169.6	43738315
	CodeSize	266448	358590	375990	384654	611139	612995	582179
	IPC	1.303540803	1.858282417	2.175421722	2.231538996	1.352498016	1.400392197	1.758279322
	Abs.Perf	1	1.159617097	1.338967025	1.376630915	1.082069609	1.025594156	1.380019627
	Impr.STI	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
STI2	Cycles	32778638.2	28315897.6	32148884	29753410.4	35006535.2	35611258	23994672.4
	Inst	44138612	54475551	67251467.2	63213376.6	42909535	45540729	43026304.6
	CodeSize	275973	373307	409083	406187	695682	697122	593202
	IPC	1.346566374	1.923850403	2.091875637	2.12457583	1.225757841	1.278829549	1.793160743
	Abs.Perf	1.047293807	1.212353049	1.067808911	1.153779158	0.980641603	0.963989107	1.430686955
	Impr.STI	4.5158%	4.3499%	-25.3939%	-19.3149%	-10.3430%	-6.3906%	3.5415%
	CS incr.	3.57%	4.10%	8.80%	5.60%	13.83%	13.72%	1.89%
STI3	Cycles	32287591	30426384.8	40136358.6	34812458.4	126383725.4	25001422.6	24182497.8
	Inst	42152936	55390134	71995391.8	64466026.6	72174460	47008437	43885393
	CodeSize	295141	400433	463889	460921	1156754	615506	612018
	IPC	1.305546022	1.820463863	1.793769896	1.851809081	0.571074003	1.880230487	1.814758482
	Abs.Perf	1.06322162	1.128259733	0.855305912	0.986108605	0.271624093	1.373076458	1.419574813
	Impr.STI	5.9462%	-2.7793%	-56.5483%	-39.6024%	-298.3703%	25.3068%	2.7864%
	CS incr.	10.77%	11.67%	23.38%	19.83%	89.28%	0.41%	5.13%

Table A.8. Performance data from DJPEG application