Procedure Cloning and Integration for Converting Parallelism from Coarse to Fine Grain

Won So and Alex Dean
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
NC State University
Raleigh, NC 27695
alex_dean@ncsu.edu

Abstract

This paper introduces a method for improving program run-time performance by gathering work in an application and executing it efficiently in an integrated thread. Our methods extend whole-program optimization by expanding the scope of the compiler through a combination of software thread integration and procedure cloning. In each experiment we integrate a frequently executed procedure with itself twice or thrice, creating two clones. Then, based on profile data we select at compile time the fastest version (original or clone) and modify call sites as needed. These techniques convert parallelism at the procedure level to the instruction level, improving performance on ILP uniprocessors. This is quite useful for media-processing applications that feature large amounts of such parallelism.

We demonstrate our technique by cloning and integrating three procedures from cjpeg and djpeg at the C source code level, compiling with four compilers for the Itanium EPIC architecture and measuring the performance with the on-chip performance measurement units. 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. For cjpeg, which is not significantly constrained by the i-cache, we find integration consistently improves code generated by all compilers but one, with a mean program speedup of 11.9%.

1 Introduction

Makers of high-performance embedded systems are increasingly using digital signal processors with VLIW and EPIC architectures to maximize processing bandwidth. However, this speed is a fraction of what it could be; it is limited by the difficulty of finding enough independent instructions to keep all of the processor's functional units busy. Extensive research is being performed to extract additional independent instructions from within a thread to increase throughput. Software thread integration can be used to merge multiple threads or procedures into one, allowing the compiler to look among multiple procedures within the application to find independent instructions and create highly efficient integrated versions. We present an approach of integrating multiple procedures in an embedded system for maximum utilization

Whole-program (or interprocedural) optimization methods extend the compiler's scope beyond the call boundaries defined by the programmer to potentially encompass the entire program. These 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. Cooper and Hall [CHK93] [Hall91] used procedure cloning to enable improved interprocedural constant propagation analysis in the matrix300 from SPEC89. Selective specialization for object-oriented languages corresponds to procedure cloning. [DC94] uses static analysis and profile data to select procedures to specialize. Procedure cloning is an alternative to inlining; a single optimized clone handles similar calls, reducing code expansion. Cloning also reduces compile time requirements for interprocedural data-flow analysis by focusing efforts on the most critical clones. Cloning is used in Trimaran [NTGR98], FIAT [CHMR95], Parascope, and SUIF [HMA95].

Region-based compilation [Hank96] allows the compiler to determine the appropriate unit of compilation, rather than being bound to the programmer-defined procedures, which are influenced more by software engineering principles than program performance. Adapting the region size as needed simplifies the tasks of optimization, register allocation and instruction scheduling. Using profile information to help guide the region selection and optimizations leads to more efficient compilation and execution. Way [Way02] extends region-based compilation to perform

inlining and cloning based on demand and run-time profile information (path spectra) rather than by default.

Software thread integration [Dean02] [Dean00] is a method for interleaving procedures implementing multiple threads of control together into a single implicitly multithreaded procedure. STI fuses entire procedures together, removing the loop-boundary constraint of loop jamming or fusion [Ersh66]. It uses code replication and other transformations as needed to reconcile control-flow differences. Only data-independent procedures are integrated; the remaining data-flow conflicts (false data dependences) are resolved through register file partitioning or register coloring. Past work used STI to interleave procedures at the assembly language level and perform fine-grain real-time instruction scheduling to share a uniprocessor more efficiently, enabling hardware-to-software migration for embedded systems. The present work uses STI to increase the instruction-level parallelism within a procedure, enabling the compiler to create a more efficient execution schedule and reduce run time.

In this paper we present methods to clone and integrate procedures to improve run-time performance and evaluate their impact. 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. We compile the programs with the cloned and integrated procedures with four different compilers (gcc, ORCC, Pro64 and Intel C++) and evaluate performance in detail.

This paper provides an initial look at the performance impact of cloning procedures in a media processing application. Although this paper describes programs in which independent procedure calls were manually identified, this is not a requirement for using the methods presented. Instead, a smart scheduler can be used in a system with multiple processes to select at run-time which procedure calls can be "shared" and executed with an efficient integrated procedure clone.

The eventual goal of this work is to automate the process of cloning and integrating procedures for VLIW/EPIC processors. We focus upon these processors because they give much more predictable performance than superscalars; this is needed for the real-time embedded applications we seek to target in future work. This present work evaluates manually transformed procedures in order to determine whether the automation would be worth the effort. We have not examined the performance impact of these methods on superscalar processors but leave this for future work.

This paper is organized as follows. Section 2 describes the techniques used to determine which procedures to clone, how to integrate them, how to modify the call sites, and then finally how to select the best clone based on performance data. Section 3 presents the experimental method: analysis and integration of the cjpeg and djpeg programs, compilers used, execution environment and profiling methods. Section 4 presents and analyzes the experimental results. Section 5 summarizes the findings.

2 Integration Methods

Planning and performing cloning and integration require several steps. We choose the candidate procedures to clone from the application and examine the applicability of STI. Then we perform integration to create the integrated clone versions of procedures and insert those in the applications. These five steps are presented in detail below.

2.1 Identifying the candidate procedures for integration

The first stage of integration is to choose the candidate procedures for integration from the application. The candidate procedures are simply those that consume a significant amount of the program's run time. These can be easily identified by profiling, which is supported by most compilers (e.g. *gprof* in gnu tools). 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) [ZVSM94]. Those routines have more loop-intensive structures and handle larger data sets, which require more memory bandwidth than normal applications. [FWL99]

2.2 System Architecture Options

Two methods for invoking integrated procedures are directly calling them in the source code or indirectly calling them by requesting a thread fork from the operating system. In both the programmer identifies work which can be parallelized

The first option speeds a single thread by integrating multiple procedure calls which can execute in parallel. The second option, when used with a multithreaded application, loosens this constraint by gathering procedure calls from separate threads.

2.3 Examining parallelism in the candidate procedure

The second step is to examine parallelism in the candidate procedure because integration requires parallel execution of procedures. Various levels of parallelism exist, from instruction to loop and procedure, based on the distribution of the independent instructions. The method proposed here is a software

technique to use STI for converting existing procedure-level parallelism to ILP. Though there are other levels of parallelism in the application we only focus on this type. Multimedia applications tend to spend most of their execution time running compute intensive routines iteratively on large independent data sets in memory. For example, FDCT/IDCT (forward/inverse discrete cosine transform), a common process in image applications, handles an 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 synchronization 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 instruction streams and can offer more ILP than the original procedure. Figure 1 shows the existing parallel threads and strategy for integration.

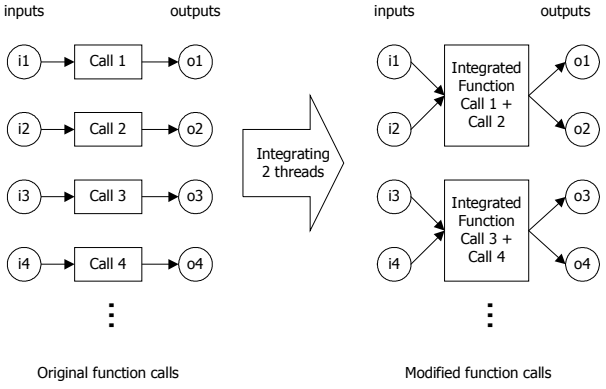

Figure 1 Parallel threads and strategy for integration

Detecting this parallelism within a single program 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 [BCGH] [Fran93] [SBV95] [Newb97] [WS91]. We are not trying to solve this problem. Instead, we assume that application developers will extract threads whether automatically or manually. We present a method to execute the parallel functions more efficiently on a single instruction stream processor. However, [So02] presents details on how to identify and group independent procedure calls within cjpeg and djpeg.

Multiple instances of a program operating on independent streams of data are independent by definition. For example, a cell-phone base station performs Viterbi decoding multiple independent data streams, as does a streaming video transcoder. To gather the work of these data streams we could rely upon a smart scheduler in the operating system to gather thread fork requests over short periods of time from multiple threads and select efficient integrated versions. This is discussed in more detail in [So02].

2.4 Performing integration

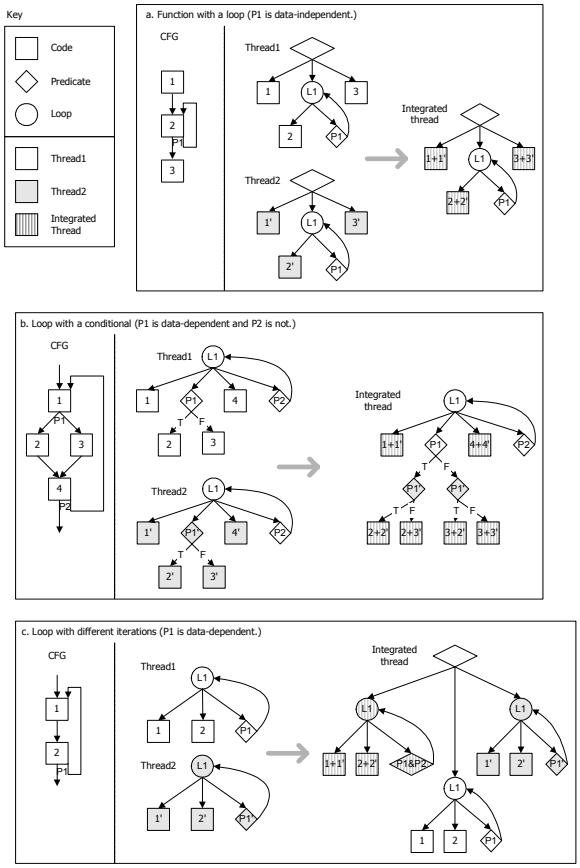

Figure 2 Code transformations for STI

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 demonstrates the cases and corresponding code transformations, which can happen integrating the same threads that handle different data sets. These transformations can be applied repeatedly and hierarchically, enabling code motion into a variety of nested control structures. This is the hierarchical (control-dependence, rather than control-flow) equivalent of a cross-product automaton. 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 loops with guarding or splitting. Finally, loops are moved into other loops through combinations of loop fusion, peeling and splitting. 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. Our past work performs assembly language level integration automatically [Dean02]. Although assembly level integration offers better control, it also requires a scheduler that targets the machine and accurately models timing. For a VLIW or EPIC architecture this is nontrivial. In this paper we integrate in C and leave scheduling and optimization to the four compilers, which have much more extensive optimization support build in. Our approach is to feed extra instruction-level parallelism to the compilers and evaluate their performance.

Whether the integration is done in 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. The second step is quite straightforward in HLL level integration because the compiler takes care of allocating registers. Not all local variables are duplicated because there may be some variables shared by the threads.

There are two expected side effects from integration. One is a code size increase (code expansion), and the other is a register pressure. The code size increases due to the multiple threads and code replication into conditionals. Code size increase has a significant impact on performance if it exceeds a threshold determined by instruction cache size and levels. The number of registers also increases approximately linearly with the number of integrated threads.

2.5 Optimizing the application

After performing integration, we have multiple versions of a specific procedure: the original discrete version and integrated versions. There are two approaches for invoking those threads in the application. The first is to modify the application to call integrated threads directly by replacing original procedure calls with integrated procedure calls. It is a static approach as it requires determining the most efficient version before compile time. The second is a dynamic approach, using a run-time mechanism to choose the most efficient version of thread at run time. Work in this area is under way. In this paper, we only focus the static approach and optimize the applications based on performance analysis.

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 procedure every time. 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 all of the calls and call 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 mispredictions.

We measure the performance of various versions of the application, varying the level of integration in the cloned procedures. We can then select the most efficient version. This grows more important if we have more than one procedure to be cloned. For example, we have three versions – original, 2-thread integrated, 3-thread integrated – of FDCT and Encode (Huffman encoding) in cjpeg application. From nine combinations to invoke those two threads, the best combination can be chosen using feedback based on the performance of each version of the thread.

2.6 Automating Integration

The goal of this work is to determine whether it is worthwhile to develop the tools needed to automatically clone and integrate procedures for high-performance embedded systems. Our previous work automatically integrates assembly language threads for Alpha and AVR architectures. Automated stages in our post-pass compiler *Thrint* include parsing, control flow and dependence analysis, data flow analysis, static timing analysis, transformation planning and execution, register reallocation and code regeneration. Supporting the currently presented work requires retargeting to support a VLIW/EPIC architecture (including parser, machine model and scheduler), a guidance layer which determines which procedures to integrate, and a lower guidance layer to control how to integrate two procedures (based upon processor utilization, code explosion, and profiling).

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 the JPEG application as presented in Section 2. Three target procedures were identified and integrated

manually at the C source code level and executed in their programs on an Itanium™ machine. The objective of the experiment is to evaluate the performance benefits and bottlenecks of STI of procedure clones.

3.1 Sample application: JPEG

JPEG is a standard image compression algorithm which is frequently used in multimedia applications. It is one of applications in MediaBench, a benchmark suite which represents multimedia workloads. [LPM97] Source code was obtained from Independent JPEG Group. We used 512x512x24bit image lena.ppm which is a standard for image compression research. JPEG is composed of two programs: djpeg (Decompress JPEG) and cjpeg (Compress JPEG). Understanding the algorithm of the application helps find existing parallelism in the application. The basic compress/decompress algorithm is presented in Figure 3. [Hals01] [Wall91]

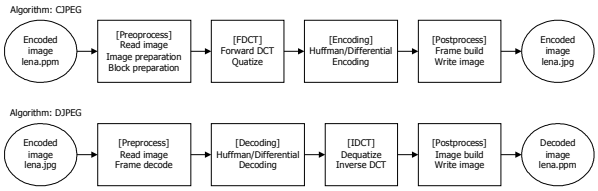

Figure 3 Algorithms of cjpeg and djpeg

3.2 Integration method

Figure 4 shows the execution time and procedure breakdown in CPU cycles from gprof compiled by GCC with –O2 optimization. djpeg spends most of its execution time performing IDCT (Inverse Discrete Cosine Transform). cjpeg also spends significant amounts of time performing FDCT (Forward DCT) and Encoding.

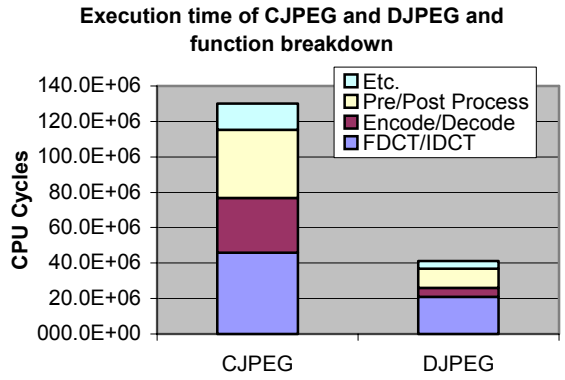

Figure 4 Execution time and procedure breakdown of cjpeg and djpeg

FDCT/IDCT is a common tool for compressing an image. The existing parallelism is that one procedure call performs FDCT/IDCT for an 8x8 pixel macro block, and input and output data of every procedure call are independent. Similarly, Encode/Decode processes a block of data with one 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 the buffer position for the blocks. We do not perform the integration for other procedures, *rgb_ycc_convert (RYC)* and *ycc_rgb_convert (YRC)* because their calls are too far apart (separated by three levels of calls), which lead to too many changes of the source code.

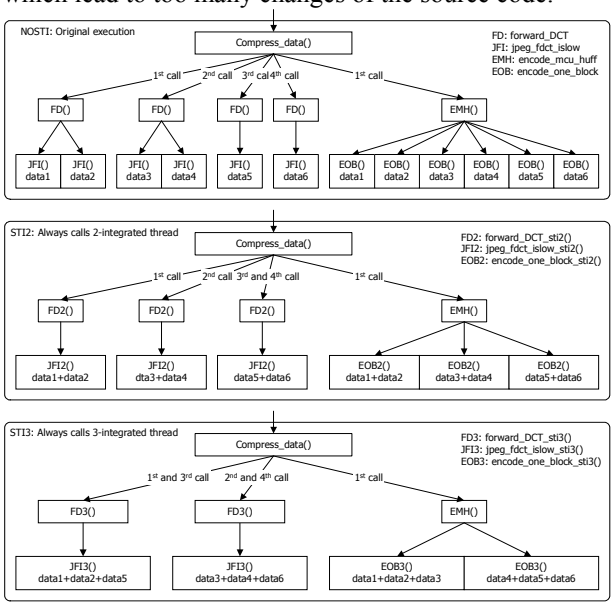

Figure 5 Execution behaviors of three versions of cjpeg application with original function, duplicated clone and triplicated clone

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) The control flow of Encode in cjpeg is also similar as it has a loop with a data-dependent 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. Even though there is a nested call from FD to JFI, the control flow is straightforward, as it is an extension of a procedure with a loop (case a in Figure 2). In this case, the intermediate procedure FD is modified to call JFI properly so that it handles the correct data sets.

We invoke the integrated threads statically by binding a specific version explicitly in the application.

Three different versions of caller procedures for the respective target procedures are written and included in the original source code, and then are compiled to different versions of application with conditional compile flags. Then we measure the performance of the target procedure with those versions. Finally, we build the best-optimized version of the application. Original execution and two implementations (STI2 and STI3) for FDCT and EOB in cjpeg are shown in Figure 5.

3.3 Overview of the Experiment and Evaluation Methods

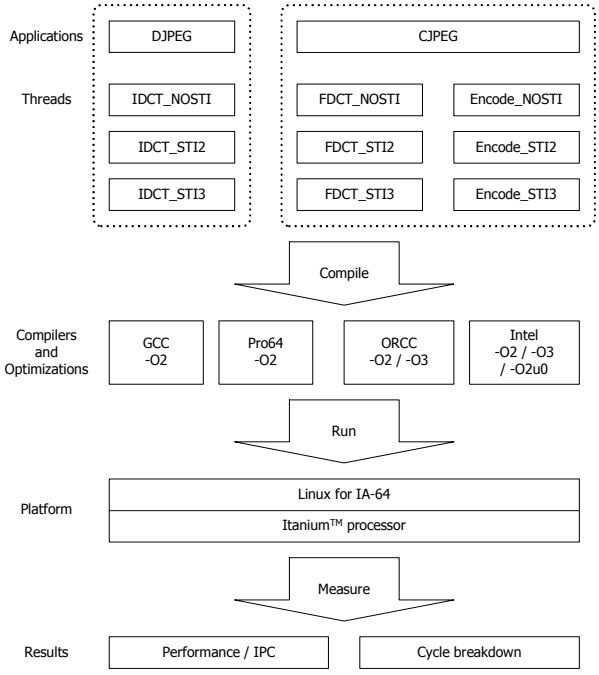

Figure 6 Overview of the experiment

Three versions of source code (1/ NOSTI: original discrete (non-integrated) version, 2/ STI2: 2-thread-integrated, 3/ STI3: 3-thread-integrated) for respective target threads (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 1 lists the compilers that we use 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 reason is that we try to observe the correlation between features of the compliers and the performance benefits of STI. Figure 6 presents the 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 1 Compilers used in the experiments

The compiled programs are executed on an Intel Itanium™ processor running Linux for IA-64. The processor features EPIC (Explicitly Parallel Instruction Computing), predication, and speculation. It can issue a maximum of 6 instructions per clock cycle and has 3 levels of caches, L1 16K data cache, L1 16K instruction cache, L2 96K unified cache, L3 2048K unified cache. It runs with 800MHz CPU clock rate and 200MHz memory bus speed. [Intel00]

All experimental data are captured during execution with 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 events. The software tool and library *pfmon* [ME02] use the PMU to measure the performance (execution cycles or time), instruction per cycle (IPC), and cycle breakdown of the procedures. All data are obtained by averaging results from 5 execution runs; there is little variation among the data.

4 Experimental Results and Analysis

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

1) CPU cycles, speedup by STI, and IPC

CPU (execution) cycles of different versions of the respective target procedures are measured and normalized compared with the performance of the original procedure compiled with GCC-O2 so that it indicates performance. Percentage performance improvement is also plotted comparing the performance of the integrated version with the original discrete one. We measure IPC for reference.

2) Cycle breakdown 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 2 shows how each category is related to the specific pipeline event. [Intel02]

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 2 Itanium™ cycle breakdown categories

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 a code size increase. We measured the pure code size of the procedure with encoded bundle size excluding data space. Code size of the application is measured with the size of the binary executable.

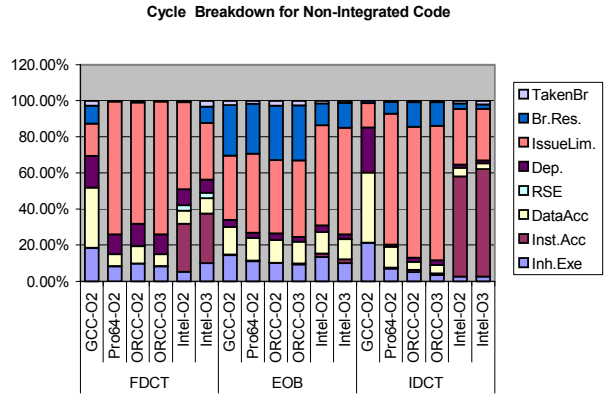

Figure 7 – Cycle breakdowns for different compilers show wide variation

Figure 7 shows the breakdown of execution cycles for each of the compilers and functions before any integration is performed. This will serve as a reference to the upcoming analysis.

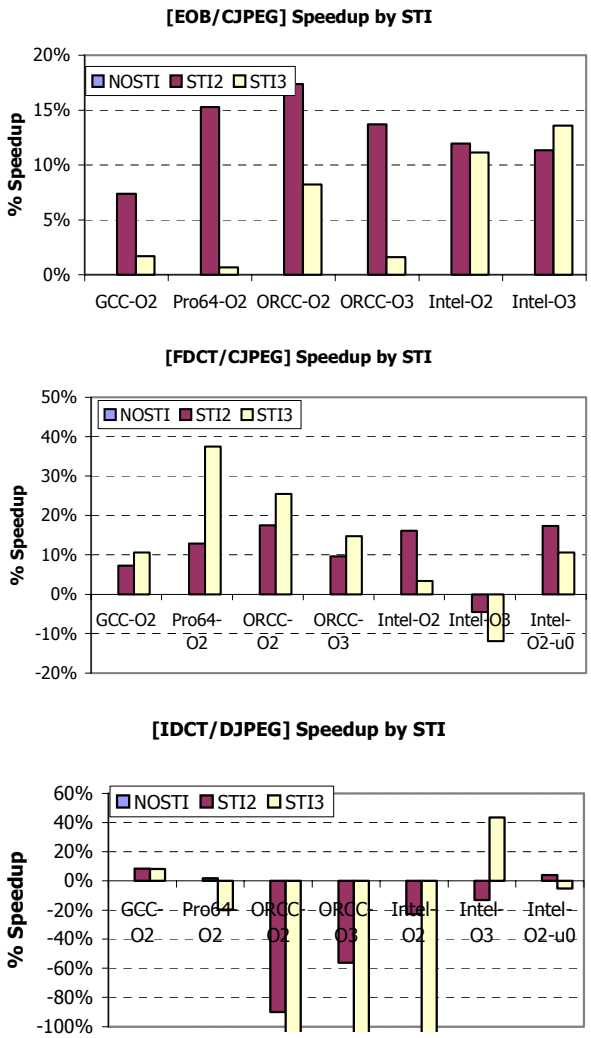

Figure 8 – Performance of integrated clones normalized to original code shows speedup and slowdown

Figure 8 shows the performance of the three integrated procedures, normalized to the performance of the original code without integration. This shows the variation in performance across all the compilers. Integrated procedure clones increase performance in all but one case for the first two experiments (FDCT and EOB). For EOB the "sweet spot" in thread count (number of procedure copies in a clone) is two, while for FDCT it is three for compilers other than the Intel compiler. The FDCT Pro64 case is interesting, as its base performance is the worst of all compilers, yet thread integration enables the compiler to bring it up to be the second best. IDCT shows a performance penalty for clone integration with ORCC and the Intel compiler, while showing a speedup for GCC and Pro64. Figure 9 shows the instructions completed per cycle for each case.

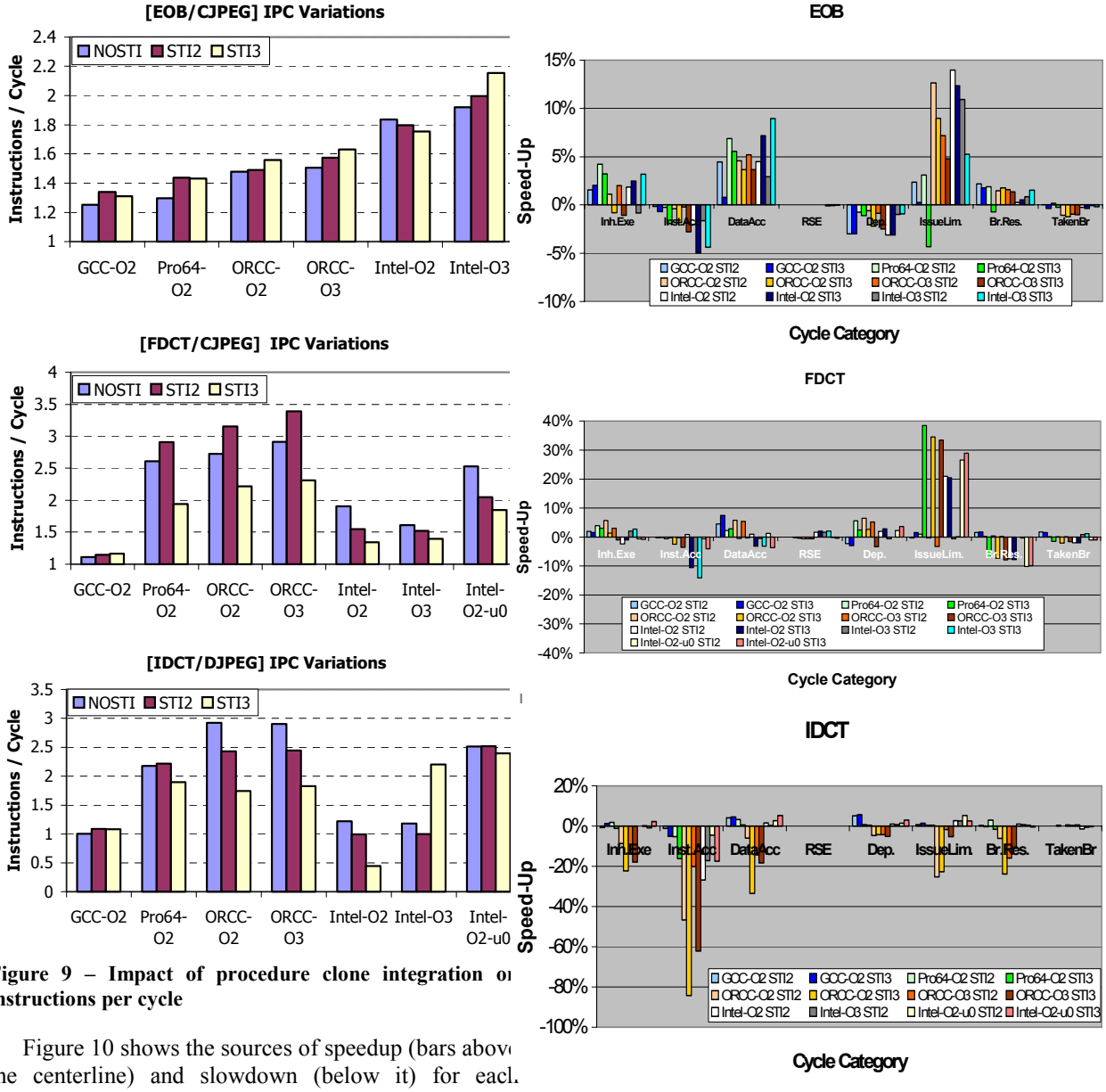

Figure 9 – Impact of procedure clone integration or instructions per cycle

Figure 10 shows the sources of speedup (bars above the centerline) and slowdown (below it) for each function. Cycle categories are listed from left to right as ordered in Table 2. Most of the performance enhancement 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 data cache access as well. The major source of slowdown for IDCT is instruction access due to code explosion, which exceeds the limits of the instruction cache (16 kbytes). In fact, code expansion limits performance for all procedures; code size should be considered when selecting procedures to clone and integrate.

Figure 10 – Breakdown of speedup/slowdown for integrated clones shows speedup comes primarily from reducing issue limited cycles and data access while slowdown comes from instruction access.

Figure 11 shows the impact of clone integration on code size. Here we see that not only does integration push IDCT up to or over the I-cache limit for all cases, but also that the Intel C++ compiler increases code size dramatically. This is due to loop unrolling. When loop unrolling is disabled for this compiler (-u0 switch), the code size becomes much more reasonable and performance recovers.

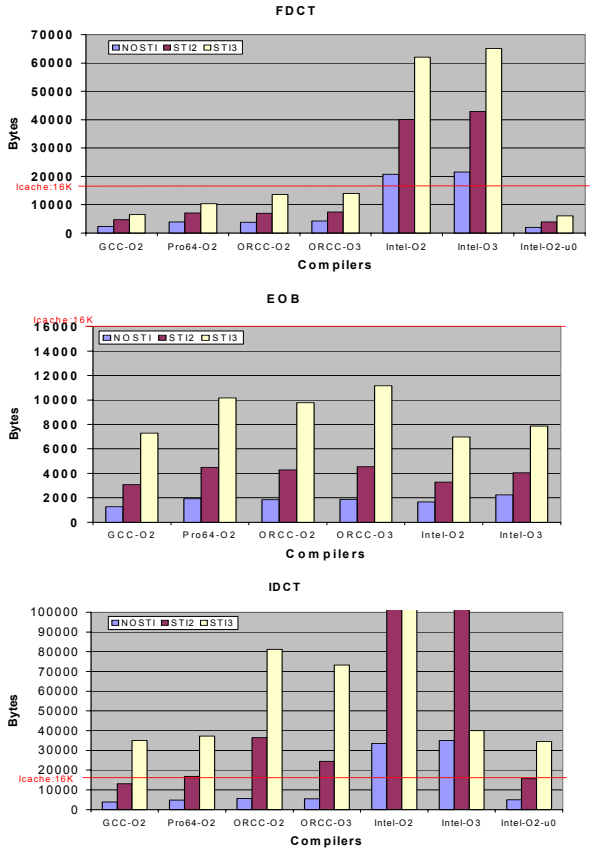

Figure 11 Impact of integration on code size compared with the 16 kByte I-Cache size

Figure 12 examines application performance for cjpeg and djpeg for various levels of clone integration for the FDCT and EOB procedures normalized to GCC –O2. The first bar indicates baseline (non-integrated) performance; the next two bars show performance for the procedures with two and three threads integrated in the clones. For the CJPEG program the "Best Combo" bar indicates performance when the best clone (with 2 or 3 threads integrated) is selected for each procedure based on measured performance, allowing the integration level to vary based on the thread count "sweet spot". It is interesting to note that integration improves performance 9% over the best compiler (intel –O2 –u0), and brings the performance of the worst (Pro64 -O2) to nearly that level (4% over intel –O2 –u0). However, the DJPEG program suffers because of the previously mentioned code expansion and limited instruction cache.

Figure 13 shows the overall cjpeg program speedup relative to the no integration case. It demonstrates that for this application, all compilers but the Intel –O3 are able to benefit from procedure cloning and integration. Also, integrating procedure clones provides nearly the same speedup to the Intel –O2 compiler, regardless of whether loop unrolling is used. The harmonic mean of speedup is 8.3% for two threads, 7.6% for three threads, and 11.9% for the best combination of thread integration levels. This shows the benefit of selecting the best number of threads per clone.

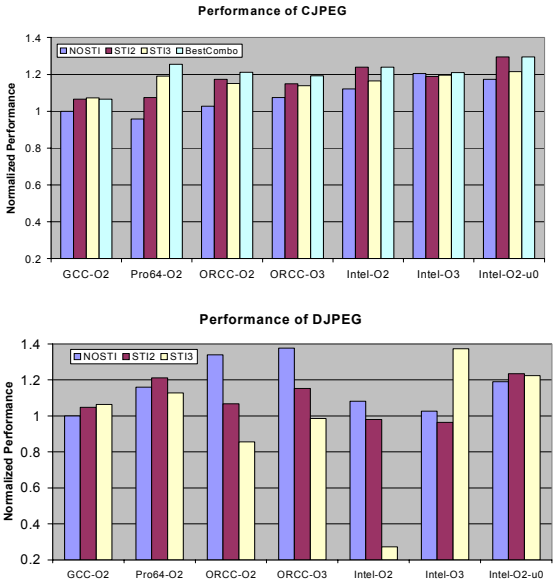

Figure 12 Overall application performance

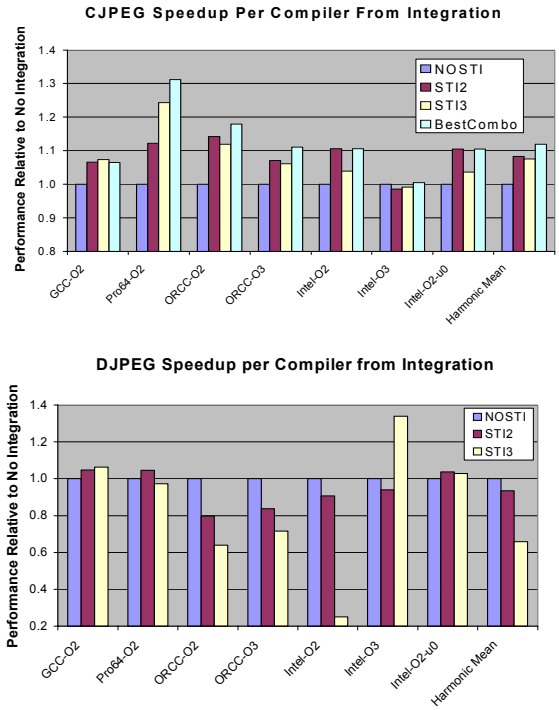

Figure 13 Overall cjpeg program performance per compiler shows integration yields speedup for nearly all compilers, yet djpeg results are mixed

Figure 12 also shows the overall speedup for djpeg. Integration provides speedups for four cases and slowdowns for three, leading to overall harmonic

means of speedups of –6.5% for two threads and –34.2% for three threads. Clearly thread integration must be applied when code expansion will not burst the instruction cache.

5 Conclusions

This paper introduces a method for improving program run-time performance by gathering work in an application and executing it efficiently in an integrated thread. Our methods extend whole-program optimization by expanding the scope of the compiler through a combination of software thread integration and procedure cloning. These techniques convert parallelism at the procedure level to the instruction level, improving performance on ILP uniprocessors. This is quite useful for media-processing applications which feature large amounts of parallelism.

We demonstrate our technique by cloning and integrating three procedures from cjpeg and djpeg at the C source code level, 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 speedups of up to 18% and program speedup up to 9%. 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.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0133690.

References

[BCGH] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV et al. "An Overview of the PARADIGM Compiler for Distributed-Memory Multicomputers" IEEE Computer 28(10)

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

[CHMR95] Alan Carle, Mary Hall, John Mellor-Crummey, Ren Rodriguez, FIAT: A Framework for Interprocedural Analysis and Transformation, Rice University CRPC-TR95522-S, 1995.

[DCG94] J. Dean, C. Chambers and D. Grove. Selective specialization for object-oriented languages. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 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.

[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. J. ACM, 13, 1 (Jan.), 17-24

[FOW87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren, "The Program Dependence Graph and Its Use in Optimization," ACM Transactions on Programming Languages, 9(3), July 1987, pp. 319-349.

[Fran93] M. Franklin. The Multiscalar Architecture. PhD Thesis, University of Wisconsin -Madison, November 1993.

[FWL99] J. Fritts, W. Wolf and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," Proceedings of SPIE, vol. 3655, pp. 2-13, Jan. 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

[Hank96] R.E. Hank, Region-Based Compilation. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996.

[HMA95] Mary W. Hall, Brian R. Murphy, and Saman P. Amarasinghe. Interprocedural analysis for parallelization: Design and experience. In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, pages 650--655. SIAM, February 1995.

[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.

[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.

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

[Newb97] Chris J. Newburn, "Exploiting Multi-Grained Parallelism for Multiple-Instruction Stream Architectures," Ph.D. Thesis, CMμART-97-04, Electrical and Computer Engineering Department, Carnegie Mellon University, November 1997

[NTGR98] A. Nene, S. Talla, B. Goldberg and R.M. Rabbah – Trimaran – an infrastructure for compiler research in instruction-level parallelism – user manual, 1998. http://www.trimaran.org. New York University

[SBV95] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. 22nd International Symposium on Computer Architecture, pp. 414–425, June 1995.

[So02] W. So, "Software Thread Integration For Converting TLP To ILP On VLIW/EPIC Architectures," Masters Thesis, Department of Electrical and Computer Engineering, NC State University, 2002

[Wall91] Gregory K. Wallace, "The JPEG Still Picture Compression standard," IEEE Transactions on Consumer Electronics, 1991.

[Way02] T.P. Way, Procedure Restructuring for Ambitious Optimization, Ph.D. thesis, University of Delaware, 2002.

[WS91] Andrew Wolfe and John P. Shen, "A variable instruction stream extension to the VLIW architecture," in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 8-11, 1991, pp. 2--14.

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