ABSTRACT

SO, WON. Software Thread Integration for Instruction Level Parallelism. (Under the direction of Associate Professor Alexander G. Dean).

Multimedia applications require a significantly higher level of performance than previous workloads of embedded systems. They have driven digital signal processor (DSP) 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, the speed is a fraction of what it could be, limited by the difficulty of finding enough independent instructions to keep all of the processor's functional units busy.

This dissertation proposes Software Thread Integration (STI) for Instruction Level Parallelism. STI is a software technique for interleaving multiple threads of control into a single implicitly multithreaded one. We use STI to improve the performance on ILP processors by merging parallel procedures into one, increasing the compiler's scope and hence allowing it to create a more efficient instruction schedule. STI is essentially procedure jamming with intraprocedural code motion transformations which allow arbitrary alignment of instructions or code regions. This alignment enables code to be moved to use available execution resources better and improve the execution schedule. Parallel procedures are identified by the programmer with either annotations in conventional procedural languages or graph analysis for stream coarse-grain dataflow programming languages.

We use the method of procedure cloning and integration for improving program run-time performance by integrating parallel procedures via STI. This defines a new way of converting parallelism at the thread level to the instruction level. With filter integration we apply STI for streaming applications, exploiting explicit coarse-grain dataflow information expressed by stream programming languages.

During integration of threads, various STI code transformations are applied in order to maximize the ILP and reconcile control flow differences between two threads. Different transformations are selectively applied according to the control structure and the ILP characteristics of the code, driven by interactions with software pipelin-

ing. This approach effectively combines ILP-improving code transformations with instruction scheduling techniques so that they complement each other. Code transformations involve code motion as well as loop transformations such as loop jamming, unrolling, splitting, and peeling.

We propose a methodology for efficiently finding the best integration scenario among all possibilities. We quantitatively estimate the performance impact of integration, allowing various integration scenarios to be compared and ranked via profitability analysis. The estimated profitability is verified and corrected by an iterative compilation approach, compensating for possible estimation inaccuracy. Our modeling methods combined with limited compilation quickly find the best integration scenario without requiring exhaustive integration. The proposed methods are automated by the STI for ILP Tool Chain targeting Texas Instrument C6x VLIW DSPs.

This work contributes to the definition of an alternative development path for DSP applications. We seek to provide efficient compilation of C or C-like languages with a small amount of additional high-level dataflow information targeting popular and practical VLIW DSP platforms, reducing the need for extensive manual C and assembly code optimization and tuning.

Software Thread Integration for Instruction Level Parallelism

by

Won So

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2007

Approved By:

Dr. Thomas M. Conte	Dr. Eric Rotenberg
Dr. Alexander G. Dean Chair of Advisory Committee	Dr. Vincent W. Freeh

To my wife and parents for their support and love ...

사랑하는 아내와 부모님께 이 논문을 바칩니다...

Biography

Won So was born in Jeonju, Korea in 1975. He grew up in Jeonju until going to Seoul National University in Seoul, Korea in 1993. After he received his Bachelor of Science degree from Seoul National University in 1997, he went to work in industry in Korea, where he developed software for plant control and monitoring systems. He began his graduate study in the Department of Electrical and Computer Engineering at North Carolina State University in 2000 and obtained his Master of Science degree in Computer Engineering in 2002.

After another year in industry where he developed software for cell phones, he went back to NCSU to pursue his Ph.D. degree in 2003. Since he began working with Dr. Alexander G. Dean in 2001, he has focused his research on software optimization techniques for embedded systems. On March 20, 2007, he successfully defended his Ph.D. dissertation.

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. It was an honor working with such a nice professor like Dr. Dean. He is a person who loves his research and enjoys what he is doing very much. His enthusiasm for this work helped me keep myself motivated and his encouragement gave me energy to continuously pursue the subject so that I could successfully finish this work. I truly appreciate his tireless support at all levels, which let me concentrate on the research comfortably.

Other committee members helped me with obtaining background and making good progress in this work. Dr. Thomas M. Conte taught me VLIW architectures and compilers, which became the target platform of this work. I learned contemporary computer architecture from Dr. Eric Rotenberg's Advanced Microarchitecture course. Dr. Vincent W. Freeh's course gave me a deep understanding about operating systems. I sincerely appreciate their invaluable advice and comments they had given to me throughout the period of my study including my qualifying review, preliminary and final oral exams. I also thank other professors in the Electrical and Computer Engineering and Computer Science departments for helping me with structuring my knowledge and developing good engineering skills.

I cannot forget the time (really the most of time while I was in Raleigh) I spent at CESR (formerly Center for Embedded System Research, now Center for Efficient, Secure and Reliable Computing). I really enjoyed occasional chats about various things, discussions about technical subjects, and taking classes and seminars together with the CESR student colleagues I have met since 2001: Zane Purvis, Balaji V. Iyer, Gregory S. Parsons, Siddhartha Shivshankar, Rony Ghattas, Karthik Sundaramoorthy, Sabina Grover, Shaolin Peng, Nagendra Kumar, Adarsh Seetharam, Yu-kuen Lai, Seongbeom Kim, Chungsoo Lim, Taemin Kim, Huiyang Zhou, Hashem Hashemi, Gary Smith, Brian Rogers, Fei Gao, and many others. The CESR lab offered a nice and friendly environment. Thanks to all these folks. I will really miss the CESR.

I would like to thank Ms. Sandy Bronson, the Administrative Assistant at CESR for being so kind to students and taking care of all necessary paperwork. I am also

thankful to the ECE graduate program director, Dr. H. Joel Trussell and other former and current ECE staff, Ms. Pascale Toussaint, Ms. Fay Bostic, and Ms. Mary Jane Johnson for helping me with going through all the administrative processes.

I would like to thank the people I met when I was doing a co-op at Atmel Corporation: Piergiovanni Bazzana, Cole VanDervort, Jim Hallman and many others. The experience at Atmel gave me a unique opportunity for hands-on experience in digital signal processing applications on real environments, which deepened my practical knowledge about this work.

I am thankful to all Korean friends I have met at NCSU since 2000: Hyoungmin Bae, Jeongbeom Ma, Youngeun Chang, Sejung Kim, Changmock Shin, Woon Chang, Dongwook Jung, Eunchu Oh, Wooyoung Choi, Jisang Hwang, Youngkyu Lee, Jinseok Park, and many others whose names are not listed here. We shared most pleasant moments like passing exams, graduations, marriages, baby births, and sporting events including NCSU football, NCAA basketball and World Cup soccer games. They also helped me with getting through the difficult times. Though most of them left Raleigh after great accomplishments in their own fields, I really felt like I made many lifelong friends here.

Many thanks also go to my friends from Seoul National University who now stay in the US, Hyungsub Choi, Inkyung Kang, Byoungyoon Kim and all other my Il-Chon's (it means the closest relative in Korean) at cyworld.com (the cyber-community service) for congratulating me and sharing this great moment with me.

I would like to thank my family. My Mom (Inkyung Yoo) and Dad (Dr. Heungyoung So) always supported me so that I could focus on my work. My sister (Yeonsook), brother (Byunghak) and their families encouraged me all the time and showed me the warmest welcome during a few short visits to Korea. I truly appreciate their help for arranging my wedding in summer 2004. I also thank my aunts and uncles living in the US for helping me with settling down here.

Most of all, I would like to express my deepest love and gratitude to my wife, Dr. Jin-Ah Park for marrying me, believing in me, giving me encouragement and love, and standing beside me always.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

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. Designers of high-performance embedded systems are increasingly using digital signal processors (DSPs) with VLIW (Very-Long Instruction Word) and EPIC (Explicitly Parallel Instruction Computing) architectures to maximize processing bandwidth while delivering predictable, repeatable performance. VLIW/EPIC processors improve the performance by issuing multiple independent instructions based on the decision of the compiler while limiting hardware complexity. Examples include Texas Instruments C6000 high performance DSPs and DaVinci Digital Media Processors [110], TriMedia media processors from NXP Semiconductors (former Philips Semiconductor) [53], TigerSHARC DSP from Analog Devices [34], ST200 (a.k.a. Lx) customizable processor families from ST Microelectronics and Hewlett-Packard [30, 52], Carmel DSP Core from Infineon Technologies [29], StarCore SC140 from StarCore [44], and MAP1000 media processor from Equator Technolo-

Table 1.1: Key features of VLIW DSPs and media processors

Processors	Issue width	Pipeline depth	Notable features
TI TMS320C62xx	8	11	1st VLIW-based DSP
TigerSHARC DSP	4	8	SIMD + VLIW data type agility
Trimedia TM1300	5	8	27 functional units and 128 registers
StarCore SC140	6	5	Scalable approach to compact code
HP/ST Lx-ST200	4	6	Scalable up to 4 clusters
Infineon Carmel DSP	2,6	8	Customizable LIW (CLIW) instructions

gies and Hitachi [38]. Table 1.1 summarizes key features of VLIW DSPs and media processors [11, 64].

The performance of VLIW/EPIC architecture processors depends on the ability of the compiler to find enough instructions which can be executed in parallel. Compilers form bigger scheduling regions using traces [32], superblocks [20, 57], hyperblocks [71] and treegions [50] to extract more independent instructions. Compilers apply aggressive instruction scheduling techniques for loops such as unrolling and software pipelining (SWP) [21, 84, 85]. However, the speed is a fraction of what it could be, limited by the difficulty of finding enough independent instructions to keep all of the processor's functional units busy.

One of the reasons is that a single instruction stream has a limited level of instruction-level parallelism (ILP) because there are not enough independent instructions within the limited size of the compiler's scheduling window [111]. 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). Many architectural designs are proposed to exploit TLP. Those include Simultaneous Multithreading (SMT) [109], Multiscalar [90], Dynamic Multithreading (DMT) [4], Welding [78] and Superthreading [108]. However, most digital signal processors do not implement these architectural extensions in order to maintain a simple architecture and predictable performance. Despite the fact that DSP and multimedia applications feature abundant thread-level parallelism, processor utilization is restricted by compilers' limited instruction scheduling windows, which

mostly do not go beyond procedure call boundaries.

1.2 Software Thread Integration for ILP

Software thread integration (STI) [27, 25, 26] is a method for interleaving threads implementing multiple threads of control together into a single implicitly multi-threaded one. Past work used STI to interleave threads at the assembly language level and perform fine-grain real-time instruction scheduling to share a uniprocessor more efficiently, enabling hardware-to-software migration (HSM) for embedded systems.

This dissertation proposes *Software Thread Integration for Instruction Level Parallelism*. We use STI to increase the instruction-level parallelism of procedures or threads, which can be seen as a superset of procedures because they may contain multiple procedures with call hierarchies. STI is used to merge multiple procedures or threads into one, effectively increasing the compiler's scope to find more independent instructions and hence allowing it to create a more efficient instruction schedule. Figure 1.1 illustrates a simple example of how STI improves performance. STI is essentially procedure jamming with intraprocedural code motion transformations which allow arbitrary alignment of instructions or code regions. This alignment is implemented by various STI code transformations which reconcile control flow differences between two procedures. It allows code to be moved to use available execution resources better and improve the instruction schedule across two procedures. Therefore, STI effectively converts existing thread-level parallelism into instruction-level parallelism, exploiting remaining idle processor resources in ILP processors.

We first investigate how to select and integrate procedures to enable conversion of coarse-grain parallelism (between procedures) to a fine-grain level (within a single procedure) using *procedure cloning and integration* [87]. These methods create specialized versions of procedures with better execution efficiency, 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

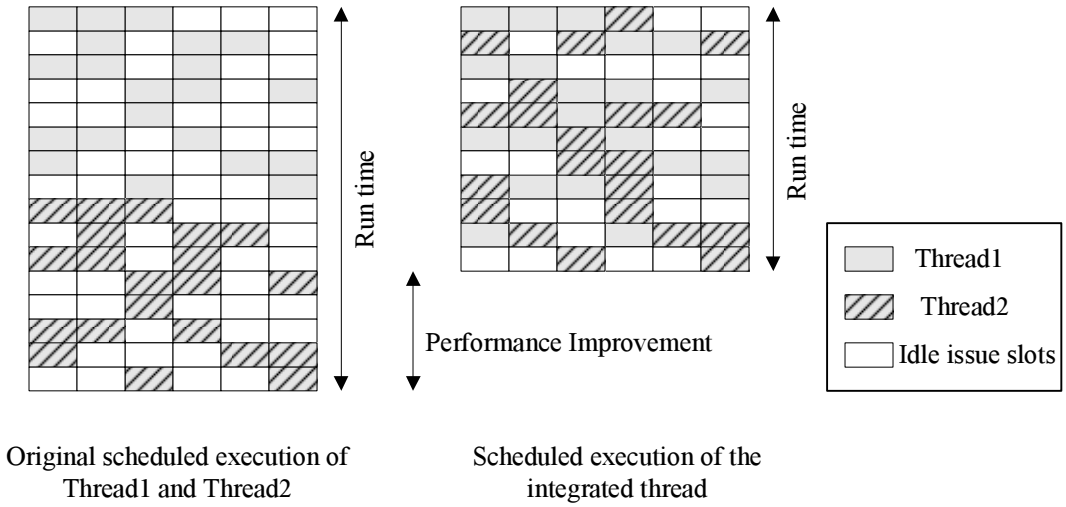


Figure 1.1: Performance benefits of STI on VLIW/EPIC machines

efficient execution by the integrated clones.

Due to difficulty of automatic extraction of independent threads by interprocedural parallelization analysis, we rely more on the programmer to identify parallel procedures. Being coarse-grain (procedure-level) dataflow representations, stream programming languages explicitly provide the data dependence information which constrains the selection of which code to integrate. Parallel procedures can come from programs written in C-like stream programming language such as StreamIt [106], which can be compiled to C for a uniprocessor. They also can come from programs written in conventional procedural languages with some language extensions (annotation or directive) added by the programmer. This is not an unrealistic scenario because abundant coarse-grain parallelism exists in media applications at the algorithm design stage but it becomes hidden after they are rewritten in sequential programming languages. We are not trying to find new parallelism but instead reclaim parallelism obscured in this process.

Rather than just merging threads, we 'integrate' them by applying various STI code transformations. These code transformations aim to maximize the ILP of the integrated thread, simultaneously reconciling control flow differences between two

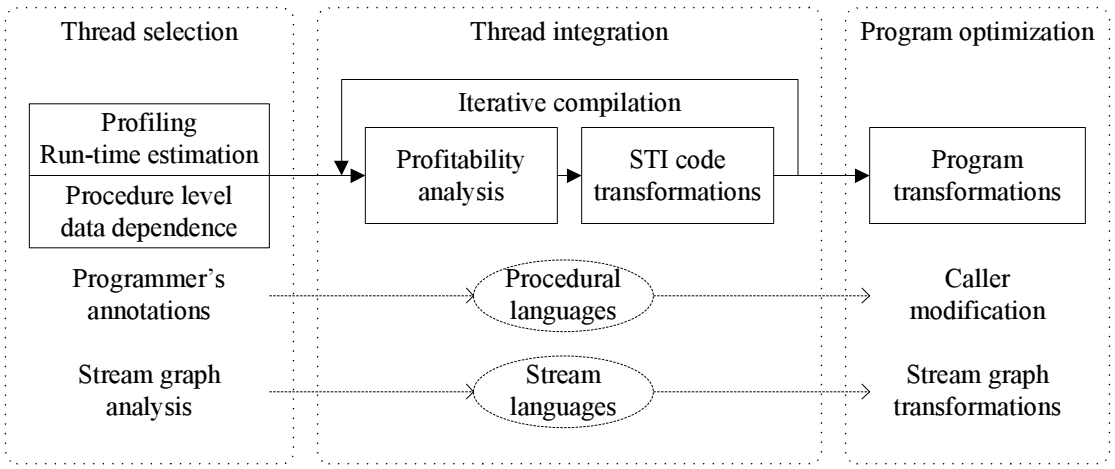


Figure 1.2: Overview of processes of STI for ILP and its application spaces

threads. In order to do this, we categorize code in terms of the control structure (cyclic or acyclic) and the instruction schedule based on the schedule analysis. Different transformations are selectively applied according to the characteristics of code to be integrated, driven by interactions with software pipelining. Code transformations involve code motion as well as loop transformations such as loop jamming, unrolling, splitting, and peeling [88].

Since STI code transformations enables arbitrary alignment of code regions between two threads, finding the best integration scenario among all possibilities without any guidance is quite exhaustive as the complexity of threads increases. In order to search and find the best integration scenario efficiently, we quantitatively estimate the performance impact of integration, allowing various integration scenarios to be compared and ranked via profitability analysis. In addition, the estimated profitability is verified and corrected by an iterative compilation approach, compensating for possible estimation inaccuracy. The iterative compilation algorithm enables quick selection of the best integrated procedure with the fewest compilation cycles [89]. The proposed methods for STI for ILP are automated by the *STI for ILP Tool Chain* targeting Texas Instrument C6x VLIW DSPs.

In this dissertation, we define and investigate the processes of STI for ILP and

explore its application spaces in detail. Figure 1.2 illustrates the processes of STI for ILP and its application spaces. STI can be used to speed up programs written in both procedural languages and stream programming languages. Procedure-level data dependence information and profile (or run-time estimation) data guide selection of threads to be integrated. Threads are integrated automatically by STI code transformations driven by profitability analysis and iterative compilation. Lastly, programs are transformed to utilize integrated threads.

Our overall goal of STI for ILP is the definition of an alternative development path for DSP applications. Currently DSP application development requires extensive manual C and assembly code optimization and tuning. We seek to provide efficient compilation of C or C-like languages with a small amount of additional high-level dataflow information (allowing the developer to leverage existing skills and code/tool base) targeting popular and practical VLIW digital signal processing platforms.

1.3 Contributions

Contributions of this dissertation are listed as follows:

- STI for ILP provides a unique way of using thread-level parallelism (TLP) more efficiently on the ILP uniprocessors with no hardware support for TLP. It enables reclaiming available execution resources in ILP processors by using additional high-level data flow information. This can offer an alternative to extensive manual performance tuning because STI for ILP automatically identifies the weak spots where the target compiler's optimization is not successful and integrates threads to exploit those existing opportunities for further optimizations.

- STI for ILP provides a comprehensive framework for applying various ILP-improving code transformation techniques consistently to maximize the overall performance of integrated thread. Rather than applying one code transformation blindly throughout the code, different code transformations are selectively

applied based on the instruction schedule analysis. It effectively combines ILP-improving code transformations with instruction scheduling techniques in order that they benefit from each other. Code transformations are driven by schedule analysis so that they complement the existing instruction scheduling techniques (software pipelining in TI C6x case). STI code transformations enable the target compiler to overcome the control flow boundaries and limited scheduling window size by scheduling the code at a coarse-grain level based on the basic-block-level instruction schedule information. In addition, STI code transformations are able to handle threads with an arbitrary control flow because transformations are applied to the hierarchical graph representations.

- We introduce a methodology for finding the best integration scenario efficiently when there are multiple possible ones. This is important because the number of possible integration scenarios increases significantly as the complexity of threads grows. Combined with quantitative estimation model for STI impact, the proposed iterative compilation approach enables quick selection of the best integration scenario, saving unnecessary compilation cycles.

- We have developed a tool chain to automate STI for ILP for research purposes. The tool chain is capable of constructing a hierarchical program representation from C source code, integrating via graph transformation, and generating the C source code for the integrated threads. Though the current implementation targets only TI C6x DSPs, the tool chain can be ported to other platforms as it integrates threads at the C source level.

1.4 Outline of Dissertation

Chapter 2 investigates the method for using STI to optimize a program written in procedural languages in detail. We present the steps for planning and performing procedure cloning and integration. The methods to select and integrate procedures, and use integrated procedures in the program are described. Threads from JPEG

applications are integrated on the Itanium processor to show benefits and bottlenecks of applying STI. This fundamental work was developed as my master's thesis [86] and is presented here as background for completeness.

Chapter 3 presents a set of STI code transformations to generate an integrated thread with better instruction schedule than original threads. We categorize the code in terms of the control structures (cyclic or acyclic) and instruction schedule based on software pipeline analysis then define various STI transformation techniques applicable in different cases. The benefits of the techniques are demonstrated with integration of various digital signal and image processing kernels on the TI C64x DSP.

Chapter 4 proposes a methodology for efficient STI to find the best integration scenario quickly. An estimation model to predict the performance impact of STI and an iterative compilation algorithm to find the best integration scenario with limited compilation cycles are presented. The effectiveness of the proposed method is demonstrated by integrating code examples from MiBench on the TI C64x DSP.

Chapter 5 describes the tool chain to automate STI for ILP. In this tool chain, the source code of the original threads is converted into a hierarchical graph representation. Based on the instruction schedule information from the assembly code compiled for a target processor (TI C6x in this case), the set of all possible integrated procedures is constructed by hierarchical graph transformation and decision tree creation. Finally, the C source code for integrated procedures is automatically generated from the transformed graph representations. The capability and utility of this tool chain are demonstrated by automatic integration of procedures from MiBench.

Chapter 6 introduces a series of methods called *filter integration*, for applying STI to streaming applications written in a stream programming language, StreamIt. Independent procedures, which correspond to parallel filters, are identified and selected for integration driven by stream graph analysis. The method for transforming the application and additional optimizations for targeting a VLIW DSP are presented. The benefit of filter integration is demonstrated by applying filter integration to a StreamIt application on the TI C67x DSP.

The rest of this chapter describes the related work.

1.5 Related Work

Instruction scheduling plays a key role in exploiting ILP for VLIW processors. STI complements various instruction scheduling techniques breaking boundaries between procedures by merging them and increasing independent instructions in basic blocks by control flow transformations and code motion. Interprocedural analysis and optimization are challenging problems for general programs. Using stream programming languages simplifies interprocedural analysis by revealing coarse-grain dependencies enabling further optimizations.

1.5.1 Instruction Scheduling

Instruction scheduling is one of the most important phases of exploiting ILP for VLIW processors. The performance depends on the ability of the compiler to find independent instructions. Since the amount of parallelism available within basic blocks tends to be quite limited, compilers performs various optimizations. Compilers perform software pipelining for cyclic code to overlap multiple iterations of a single loop in software. For acyclic code, compilers form larger scheduling region by merging multiple basic blocks.

Software pipelining

Software Pipelining (SWP) [21, 84, 85] is a scheduling method to run different iterations of the same loop in parallel. It finds independent instructions from the following iterations of the loop and typically achieves a better schedule. However, software pipelining can suffer or fail when confronted with complex control flow, excessive register pressure, or tight loop-carried dependences. Overcoming these limitations has been an active area of research for over a decade. Much work has been performed to make software pipelining perform better on code with multiple control-flow paths, as this presents a bottleneck.

Hierarchical reduction merges conditional constructs into pseudo-operations and

list schedules both conditional paths. The maximum resource use along each path is used when performing modulo scheduling. The pseudo-operations are then expanded and common code is replicated and scheduled into both paths [66]. If-conversion converts control dependences into data dependences. It guards instructions in conditionally executed paths with predicate conditions and then schedules them, removing the conditional branch [6]. Enhanced modulo scheduling begins with if-conversion to enable modulo scheduling. It then renames overlapping register lifetimes and finally performs reverse if-conversion to replace predicate define instructions with conditional branch instructions [112, 113], eliminating the need for architectural predication support.

These approaches schedule all control-flow paths, potentially limiting performance due to the excessive use of resources or presence of long dependence chains. Lavery developed methods to apply modulo scheduling to loops with multiple control-flow paths and multiple exits. Speculation eliminates control dependences and creation of a superblock or hyperblock removes undesirable paths from the loop based upon profile information. Epilogs are created to support the multiple exits from the new loop body [68].

The previous methods require that one initiation interval (II) is used for all different paths. Some research focuses on the use of path-specific IIs to improve performance. In GURPR each path is pipelined using URPR and then code is generated to provide transitions between paths. However, this transition code is a bottleneck [96]. All-paths pipelining builds upon GURPR and perfect pipelining [3], by first scheduling the loop kernel along execution path. A composite schedule is then generated by combining the path kernels with code to control the switching among them [92]. Code expansion can be a problem due to the exponential increase of potential paths. Modulo scheduling with Multiple IIs relies upon predication and profiling information [114]. After if-conversion, it schedules the most likely path and then incrementally adds less likely basic blocks from conditionals.

Pillai uses an iterative approach to move instructions out of conditionals for a clustered VLIW/EPIC architecture. The compiler repeats a sequence: speculation to expose more independent instructions, binding instructions to machine clusters,

modulo scheduling the resulting code, and saving the best schedule found so far [79].

Our use of STI to improve the performance SWP for control-intensive loop is different from existing work. Integration increases the number of independent instructions visible to the compiler while effectively reducing loop overhead (through loop jamming). This enables the compiler to use existing SWP methods to create more efficient schedules.

Global acyclic scheduling

Compilers try to extract more ILP by forming bigger scheduling regions for acyclic code through merging multiple basic blocks. In trace scheduling [32], compilation proceeds by selecting a likely path of execution, called a trace. This trace is compacted by performing code motion without regard to basic block boundaries. Correct execution is preserved by off-trace compensation code. Superblock scheduling [20, 57] forms superblocks, regions with a single entrance and (possibly) multiple exits. Side entrances to superblocks are eliminated by tail duplication. Both of techniques use profile information to guide the region formation process, so that the most often executed paths reap the greatest benefit. A hyperblock [71] is a set of predicated basic blocks, in which control may only enter from the top, but may exit from one or more locations. It uses tail duplication to remove side entrances as superblock scheduling and if-conversion to replace a set of basic blocks containing conditional control flow with a single block of predicated instructions. Treegion scheduling [50] forms a treegion, single-entry/multiple-exit global scheduling region which consists of basic blocks with control flow forming a tree. Instructions moved above a branch are speculated and register renaming is used to prevent violation of program semantics.

In region scheduling [41, 8], a program is divided into regions containing statements requiring the same control conditions via the Program Dependence Graph (PDG). Guided by the estimates of the parallelism present in the program regions, the region scheduler repeatedly transforms the PDG, uncovering potential parallelism in regions. The transformations, which include loop unrolling and invariant code motion, forward/backward code motion, and region copying/collapsing, redistribute

parallelism among regions through the transfer of code from one region to another.

Different from these techniques, STI assists the compilers to exploit parallelism across different procedures by merging them. STI also allows arbitrary alignment of instructions or code by control flow transformations and code motion so that compilers use independent instructions efficiently.

1.5.2 Loop Optimizations

Loop jamming and unrolling

Loop jamming (or fusion) and unrolling are well-known optimizations for reducing loop overhead. Unroll-and-jam [5, 2, 15, 14] can increase the parallelism of an innermost loop in a loop nest. This is especially useful for software pipelining as it exposes more independent instructions, allowing creation of a more efficient schedule [18]. Unrolling factors have been determined analytically [17].

Loop jamming, unrolling, and unroll-and-jam have been used to distribute independent instructions across clusters in a VLIW architecture to minimize the impact of the inter-cluster communication delays [82, 83]. The technique called Deep Jam also uses loop jamming to increase ILP [19]. It recursively applies combinations of loop jamming and code duplication for inner loops and conditionals. However, transformation is limited to threads (threadlets) with the identical control flow while STI works both identical and heterogeneous control structures.

STI is different from these loop-oriented transformations in two ways. First, STI merges separate procedures, increasing the number of independent instructions within the compiler's scope. Second, STI distributes instructions or code regions to locations with idle resources, not just within loops. It does this with code motion as well as loop transformations (peeling, unrolling, splitting, and jamming).

1.5.3 Interprocedural Optimization and Analysis

Interprocedural (or whole-program) optimization methods extend the compiler's scope beyond the procedure boundaries defined by the programmer to potentially encompass the entire program. These methods include procedure inlining and procedure cloning. STI for ILP can take advantage of interprocedural parallelization analysis if it successfully extracts parallel threads from sequential programs.

Procedure inlining

Procedure inlining (inline expansion or inline substitution) is a widely researched and accepted means to enable whole program optimization. It improves compiler analysis and enables other optimizations as well as instruction scheduling [7, 56, 24]. Way [116, 115] extends region-based compilation [49] to perform inlining and cloning based on demand and run-time profile information.

STI differs with procedure inlining techniques in two ways. First, overlap of the code of independent procedures by STI is not limited to the callees of the same procedure. Second, STI does not only merge two procedures but also transforms control structures so that the compiler generates a more efficient schedule.

Procedure cloning

STI leverages procedure cloning, which 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 [23, 45] 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. Static analysis and profile data have been used to select procedures to specialize [28]. 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 [75], FIAT [47], Parascope [16], and SUIF [48].

Interprocedural parallelization analysis

There are numerous research compilers which support automatic parallelization. Those include Parafrase-2 from University of Illinois [80], ParaScope from the Rice University [22], Polaris from the University of Illinois [13], and the SUIF compiler from Stanford University [46]. These compilers automatically parallelize loops in programs written in Fortran or C by incorporating variety of analyses such as scalar and array analyses, symbolic analysis, and privatization analysis. Parafrase-2, ParaScope, and SUIF are also capable of interprocedural data flow analysis, where its analyses are performed across procedure boundaries enabling parallelization of coarse-grain loops. (See [72] for a detailed survey.)

These compilers are noticeably successful in parallelizing scientific and numeric applications composed of counted loops with very regular and analyzable structures, and a predictable array access pattern [61]. However, complier writers have had little success in extracting thread-level parallelism from sequential programs for generic purposes, which may include complex control flow, pointer accesses, and recursive data structures [77].

If these parallelizing compilers are able to extract parallel procedures (or threads) from DSP or multimedia applications, STI for ILP can leverage them by integrating those threads. However, the chances are limited because these compilers focuses on finding parallelism only from loops (especially outer loops from perfectly nested loops) while parallel procedures often reside in a non-loop form (e.g. sequential calls of different procedures) in real applications. Though multimedia applications show relatively simpler pointer behaviors than general programs, the interprocedural pointer analysis is another difficulty because most procedure calls involve passing input or output pointers. (Hind et al. [51] describes the comparison of different pointer analysis algorithms and Hunter et al. [55] evaluates them for media applications.)

1.5.4 Stream Programming

Rather than performing complex interprocedural analysis we rely upon finding parallelism explicit in a higher-level stream program representation. For DSP applications written in a programming language such as C, opportunities for optimizations beyond procedure level are hidden and hard for compilers to recognize. A stream program representation makes data independence explicit, simplifying the use of our methods to improve performance.

Stream-based programming dates to the 1950s; Stephens provides a survey of programming languages supporting the stream concept [91]. LUSTRE [43] and ESTEREL [12] are common synchronous dataflow languages. Performing signal processing involves using a synchronous deterministic network with unidirectional channels. The SIGNAL language was designed for programming real-time signal processing systems with synchronous dataflow [37].

Recent work has focused on two fronts: improving the languages to make them more practical (adding needed features and making them easier to compile to efficient code) and developing multiprocessors which can execute the stream programs quickly and efficiently. Khailany et al. introduced Imagine, composed of a programming model, software tools and stream processor architecture [62]. The programming language and the software tools target the Imagine architecture. Thies et al. proposed the StreamIt language and developed a compiler [106]. The StreamIt language and compiler have been used for the Raw architecture [97], VIRAM [74] and Imagine [62], but they also can be used for more generic architectures by generating C code for a uniprocessor.

Overview of StreamIt

StreamIt [39] programs consist of C-like filter functions which communicate using queues and global variables. Apart from the global variables, the program is essentially a high-level dataflow graph. There are *init* and *work* functions inside filters, which contain code for initiation and execution respectively. In the *work* function, the

filter can communicate with adjacent filters with *push(value)*, *pop()*, and *peek(index)*, where *peek* returns a value without dequeuing the item.

Filters can be connected with three constructs: *Pipeline*, *SplitJoin* and *FeedbackLoop*. The output of the first filter (or one of other constructs) is connected to the input of the second filter by the *Pipeline* construct. *SplitJoin* is used to specify parallel filters that diverge from a common *splitter* and merge into a common *joiner*. There are two types of splitter: *duplicate* and *roundrobin*. A *FeedbackLoop* is used to create a cycle between filters. Stream applications are designed conceptually with these constructs and written in the StreamIt language [105].

A program written in the StreamIt language can be compiled for the Raw architecture or general uniprocessors by the StreamIt compiler. The compiler translates StreamIt syntax and constructs the StreamIt Intermediate Representation (SIR) that encapsulates the hierarchical stream graph. Since filters have different data rate and scope, as defined by number of *pop*, *push* and *peek*, filters are scheduled to work synchronously [60]. Based upon this schedule, the necessary buffers between filters are allocated by the compiler. There are two types of schedule: the steady state schedule and the initialization schedule. From a performance standpoint, the steady state schedule dominates, as initialization occurs only once.

The back-end processes of the StreamIt compiler depends on the target. For Raw, it automatically estimates work size of filters and partitions the stream graph to balance the processing. Then it generates C code for each tile (the processing unit of Raw) by merging (*fusion*) or separating (*fission*) filters. For uniprocessors, the compiler simply generates a C program which can be compiled with any C compilers.

The StreamIt program representation provides a good platform for applying STI for ILP. Most importantly, it fully exposes parallelism between filters. The arcs in the stream graph defines use-def chains of data. Thus, the filters which are not linked each other directly or indirectly do not depend on each other. Since a single filter in a StreamIt program is converted into a single procedure in the C program, the granularity level of parallelism expressed in a StreamIt program matches what is needed for STI.

StreamIt optimizations

There have been many optimizations developed for StreamIt programs. The most relevant one to our work is fusion transformation, which joins multiple filters into one. There are two types of fusion transformation: vertical fusion of *Pipeline* constructs and horizontal fusion of parallel filters between *SplitJoin* constructs [39, 107].

Vertical fusion is not immediately of interest since its advantage mainly comes from converting channels between filters to local buffers [81]. For a VLIW uniprocessor, our target architecture, localizing communication is not as critical to performance as it is for communication-exposed architectures. These architectures depend upon load balancing and minimizing communication for efficient execution while we focus on finding opportunities to run multiple filters concurrently.

Horizontal fusion performs fusion of multiple parallel filters located between *SplitJoin* constructs. An algorithm is presented by Gordon et al. [39], while the requisite queue access (*push*, *pop*, and *peek*) modifications are showed by Theis et al. [107]. After fusion, however, the code for parallel filters in one fused filter are not scheduled to run explicitly in parallel. The generated code for each filter is concatenated so that it execute two independent parallel filters serially. Thus further optimization opportunities are left for the target compiler. If the target compiler can not detect independent operations, opportunities for further optimizations are lost. This is often the case when the two fused filters have loops, in which case the compiler must be able to perform inter-loop code motion or even loop fusion.

Unlike filter fusion, our method uses data dependence information expressed explicitly in the stream programming language to guide and perform *filter integration*. It applies various STI code transformation techniques so that it achieves more fine-grain schedules by interleaving code from two independent filters.

Linear optimizations [67] also can result in collapsing of filter functions, with speedups of 450% reported for a benchmark suite of DSP applications. However, this is possible only for a limited set of filter functions: those for which each of the outputs can be expressed as an affine combination of its inputs, such as FIR filters, expanders, compressors, FFTs and DCTs. Our method does not have such a

constraint, and hence is more general.

Chapter 2

Procedure Cloning and Integration

2.1 Introduction

Software thread integration (STI) 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. In this chapter, we present an approach of integrating multiple procedures in an embedded system for maximum processor utilization. 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.

Although this chapter describes programs in which independent procedure calls were manually identified, this is not a requirement for using the methods presented. Original programs can be multithreaded using thread library or parallel programming directives like OpenMP. Parallel procedures can be identified via language extensions (e.g. annotation or directive) by programmers before integration. They can come from

programs written in C-like stream programming language such as StreamIt. Parallel procedures even can be found by interprocedural parallelization analysis though it is difficult in reality.

This chapter presents the foundations of STI for ILP, developed in my master's thesis [86], and is included as background for completeness. Section 2.2 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 2.3 presents the experimental method: analysis and integration of the *cjpeg* and *djpeg* programs, compilers used, execution environment and profiling methods. Section 2.4 presents and analyzes the experimental results. Section 2.5 summarizes the findings.

2.2 Methods

Planning and performing procedure 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 procedure integration to create the integrated clone versions of procedures and insert those in the applications. These steps are presented in detail below.

2.2.1 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 (e.g. FIR, IIR), and frequency-time transformations (e.g. FFT, DCT) [119]. Those routines have more loop-intensive structures and handle larger data sets, which require more

memory bandwidth than normal applications [35].

2.2.2 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 2.1 shows the existing parallel procedure calls 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 [10, 33, 90, 76, 117]. 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.

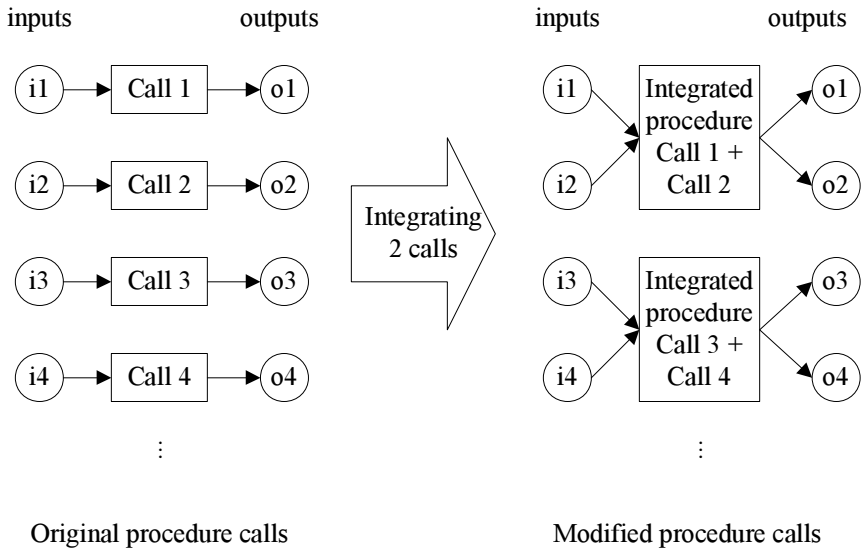


Figure 2.1: Parallel procedure calls and strategy for STI

2.2.3 Performing Procedure Integration

Many cases and corresponding code transform techniques for STI have been already demonstrated by previous work [27, 25, 26]. STI uses the control dependence graph (CDG, a subset of the program dependence graph) [31] 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; Integration of procedures with loops involves jamming two loops. To move code into a conditional it is replicated into each case. Various code transformations are applied depending on the control flow and the ILP characteristics. STI code transformation techniques for ILP are mainly discussed in Chapter 3.

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 [26]. 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 work, we integrate in C and leave scheduling and optimization to the compilers, which have much more extensive optimization support build in.

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 operations and local variables are duplicated because there may be some operations and variables shared by the threads. For example, a loop induction variable and its increment operation are not duplicated when loops with fixed iteration counts are jammed.

There are three expected side effects from integration: increases in code size, register pressure, and data memory traffic. The code size increases due to code copy and replication introduced by code transformations. Code size increase has a significant impact on performance if it exceeds a threshold determined by instruction cache size(s). The register pressure also increases with the number of integrated threads and can lead to extra spill and fill code, reducing performance. Finally, the additional data memory traffic may lead to additional cache misses due to conflicts or limited capacity.

2.2.4 Optimizing the Application

After performing procedure integration, we have multiple versions of a specific procedure: the original discrete version and integrated versions. There are two approaches for invoking those procedures in the application. The first is to modify the application to call integrated procedures 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. In this work, we only focus the static approach and optimize the applications based on performance analysis. A dynamic approach will be investigated in future work.

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. Hence control flow becomes slightly more complicated than before. As a result, some overhead is unavoidable from register pressure and branch mispredictions.

Base on the information from static analysis or profiling, the most efficient version of the procedure is selected. 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 procedures, the best combination can be chosen using feedback based on the performance of each version of the procedure.

2.3 Experiments

We chose the JPEG application as an example of the multimedia workload. We performed procedure cloning and integration for the JPEG application. Three target procedures were identified and integrated manually at the C source code level. The code was compiled by four different compilers with various optimization flags. The compiled programs were executed on an Itanium machine.

2.3.1 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 [69]. 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).

2.3.2 Integration Method

The execution profile from *gprof* shows that *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.

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 (*JII: jpeg_idct_islow*) in *djpeg*, FDCT (*FD: forward_DCT*, *JFI: jpeg_fdct_islow*) and Encode (*EOB: encode_one_block*) in *cjpeg*. Decode (*DM: decode_mcu*) in *djpeg* cannot be parallelized because of the data dependencies between the buffer positions for the blocks.

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. The control flow of Encode in *cjpeg* is also similar as it has a loop with a data-dependent predicate. 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. 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 procedures 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

Table 2.1: Compilers used in experiments

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

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.

2.3.3 Overview of Experiments and Evaluation Methods

Three versions of source code (*NOSTI*: original discrete (non-integrated) version, *STI2*: 2-thread-integrated, *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 –O2-u0 (–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 2.1 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.

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

Table 2.2: Itanium cycle breakdown categories

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 oversubscription or asymmetries
Br. Res. (Branch re-steers)	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

memory bus speed [58]. 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* [73] 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.

2.4 Experimental Results and Analysis

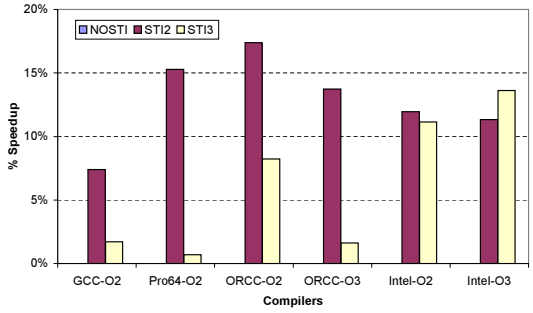
Two kinds of data are obtained to observe the performance and execution behavior of the integrated threads.

Speedup by STI CPU (execution) cycles of different versions of the respective target procedures are measured. Percentage performance improvement (speedup) is plotted comparing the performance of the integrated version with the original discrete one.

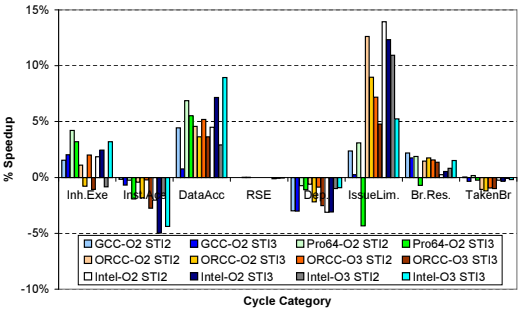
Speedup breakdown Every cycle spent on running program on Itanium can be separated in two categories: The first is an 'inherent execution cycle', a cycle used to do the real work of the program and the other is a 'stall', the cycles lost waiting for a hardware resource to become available. The stall can be also subdivided to seven categories: data access, dependencies, RSE activities, issue limit, instruction access, branch re-steers, and taken branches. Table 2.2 shows how each category is related to the specific pipeline event [59]. 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.

Figure 2.2 shows the speedup of the three integrated procedures over 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. IDCT shows a performance penalty for integration with ORCC and the Intel compiler, while showing a speedup for GCC and Pro64.

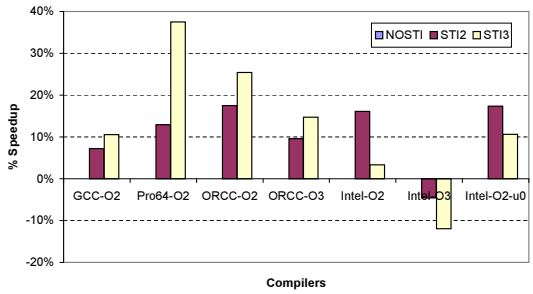
Figure 2.3 shows the sources of speedup (bars above the centerline) and slowdown (below it) for each function. Cycle categories are listed from left to right as ordered in Table 2.2. Most of the performance enhancement results from reducing issue-limited cycles, showing how the compilers are able to generate better schedules when given more independent instructions. Some improvement comes from data cache access



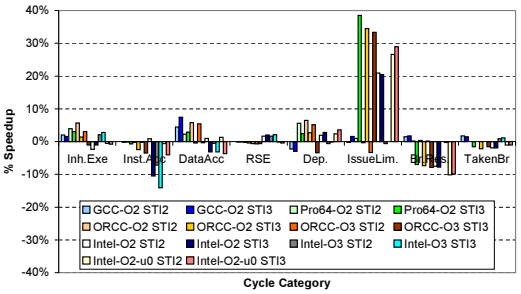
(a) EOB/CJPEG



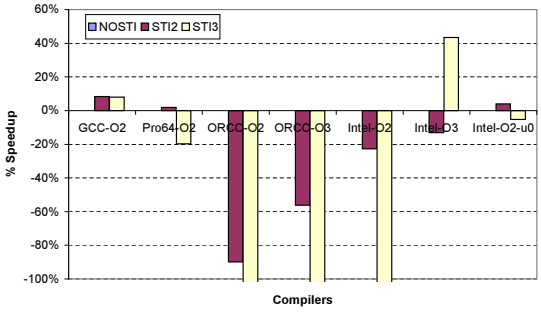
(a) EOB/CJPEG



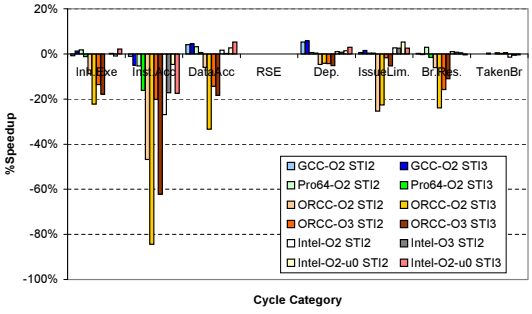
(b) FDCT/CJPEG



(b) FDCT/CJPEG



(c) IDCT/DJPEG



(c) IDCT/DJPEG

Figure 2.2: Speedup of integrated clones

Figure 2.3: Speedup breakdown

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.

Overall *cjpeg* program speedup reaches up to 11% over the best compiler (intel-O2-u0) by selecting the best combination of the integrated clones. However, the *djpeg* program suffers because of the previously mentioned code expansion and limited instruction cache.

2.5 Conclusions

This chapter overviews 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 17% 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 STI.

Chapter 3

STI Code Transformations for ILP

3.1 Introduction

Code transformation plays the most important role in software thread integration (STI) for instruction level parallelism (ILP) because improvement of ILP comes from moving instructions into the regions with idle execution resources. This chapter presents a set of STI code transformation techniques to improve ILP when integrating two procedures (or threads) on VLIW processors.

To achieve this goal, we have to consider two requirements. The first is that different code transformations should be applied depending on the original code's instruction schedule because the instruction schedule is not uniform throughout the code. The instruction schedule of integrated code depends on how effectively we move the instructions in one thread to the other with available resources. The second is that code transformations should complement existing instruction scheduling techniques. Software pipelining (SWP) is a critical optimization for producing efficient instruction schedule for VLIW processors in high-performance embedded applications such as digital signal processing. However, software pipelining can suffer or fail when confronted with complex control flow, excessive register pressure, or tight loop-carried

dependences. In order to complement SWP, STI code transformations should take advantage of cases where SWP performs poorly or fails.

In this chapter, we present the classification of code in terms of the control structure and instruction schedule, based on the software pipelining analysis. We propose a method to determine when and by which transformations we integrate procedures depending on the ILP characteristics of code to be integrated, driven by interactions with software pipelining. We then show that the proposed STI code transformations significantly improve the instruction schedule and run-time performance of the code, where software pipelining performs poorly or fails by integrating procedures from Texas Instrument (TI) DSP/Image Libraries and executing them on a TI TMS320C64x VLIW DSP.

This chapter is organized as follows. Section 3.2 describes our methods for analyzing code and performing integration. Section 3.3 presents the hardware and software characteristics of the experiments run, which are analyzed in Section 3.4.

3.2 Methods

Software pipelining is a critical optimization which can dramatically improve the performance of loops and hence applications which are dominated by them. However, software pipelining can suffer or fail when confronted with complex control flow, excessive register pressure, or tight loop-carried dependences. Software thread integration (STI) can often improve the performance of looping code in cases where software pipelining performs poorly or fails. In this section, we present methods to apply STI for improving the performance of looping code on a VLIW DSP.

3.2.1 Classification of Code

To examine the effects of SWP in a VLIW DSP, we investigate the schedules of procedures from TI DSP and Image/Video Processing Libraries by compiling them with the C6x compiler for the C64x platform [101, 102]. Among 92 inner-most loops in

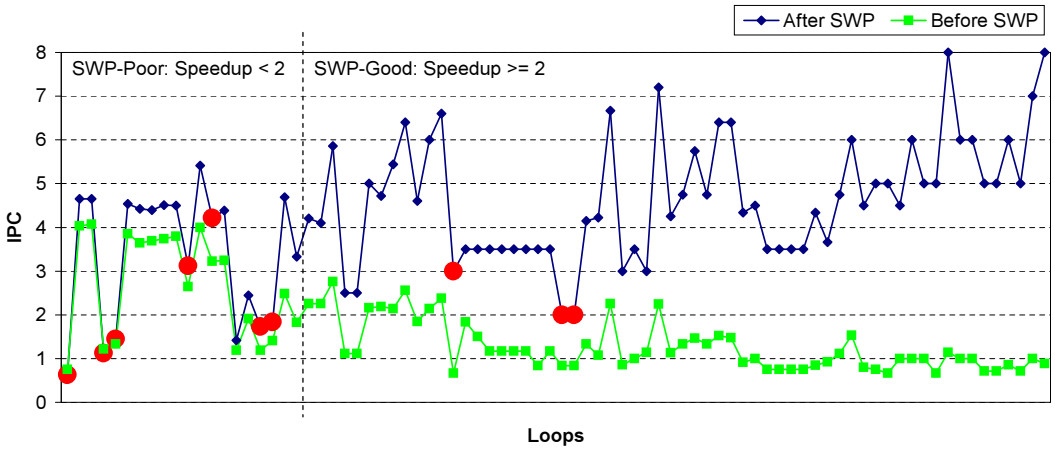


Figure 3.1: IPC of software pipelined loop kernels

68 procedures, 82 are software pipelined by the compiler. Figure 3.1 shows instruction per cycle (IPC) of the loop kernels before and after SWP. Loops are ordered by increasing speedup by SWP. The vertical dotted line shows the boundary between SWP-Good and SWP-Poor loops. The loops with large red circles are *dependence bounded*.

Based on these measurements, we classify code in three categories based upon the impact of software pipelining:

SWP-Good code benefits significantly from software pipelining, with initiation interval (II) improvements of two or more. IPCs of these loop kernels are mostly larger than 4.

SWP-Poor code is sped up by a factor of less than two using software pipelining. The IPCs of these loop kernels mostly remain less than 4, except for the loops which already had high IPCs before SWP.

SWP-Fail code causes attempts to software pipeline to fail.

Analysis of the pipelined loop kernels in the SWP-Poor category shows that IPCs are low if Minimum Initiation Interval (MII) is bounded by Recurrence MII (RecMII) rather than Resource MII (ResMII). These are called *dependence bounded* loops. The

loops are *resource bounded* otherwise. In Figure 3.1, ten loops (emphasized with the large red circles) are dependence bounded loops, and generally are SWP-Poor and low-IPC loops.

There are various reasons for SWP-Fail loops: (1) A loop contains a call which can not be inlined, such as a library call. (2) A loop contains a control code which can not be handled by predication. (3) There are not enough registers for pipelining loops because pipelined loops use more registers by overlapping multiple iterations. (4) No valid schedule can be found because of resource and recurrence restrictions.

3.2.2 Applying STI to Loops

Our goal in performing STI for processors with support for parallel instruction execution is to move code regions to provide more independent instructions, allowing the compiler to generate a better schedule. As loops often dominate procedure execution time, STI must distribute and overlap loop iterations to meet this goal.

In STI, multiple separate loops are dealt with using combinations of loop jamming (fusion), loop splitting, loop unrolling and loop peeling. The characteristics of both the loop body and the surrounding code determine which methods to use. Figure 3.2 illustrates representative examples of code transformations for loops. It shows control flow graphs of original and integrated procedures before and after STI code transformations for loops: (a) Loop jamming + loop splitting, (b) Loop unrolling + loop jamming + loop splitting, (c) Loop peeling + loop jamming + loop splitting.

loop jamming+splitting works by jamming both loop bodies then leaving the original loops as 'clean-up' copies for remaining iterations. This is appropriate when both loop bodies have low IPC. The jammed loop has a better schedule than the original loops.

loop unrolling+jamming+splitting works by unrolling one loop then fusing two loop bodies. This transformation is beneficial when two loops are asymmetric in terms of size as well as IPC. The best unroll factor is approximated by considering both iteration counts and instruction schedules of the loops.

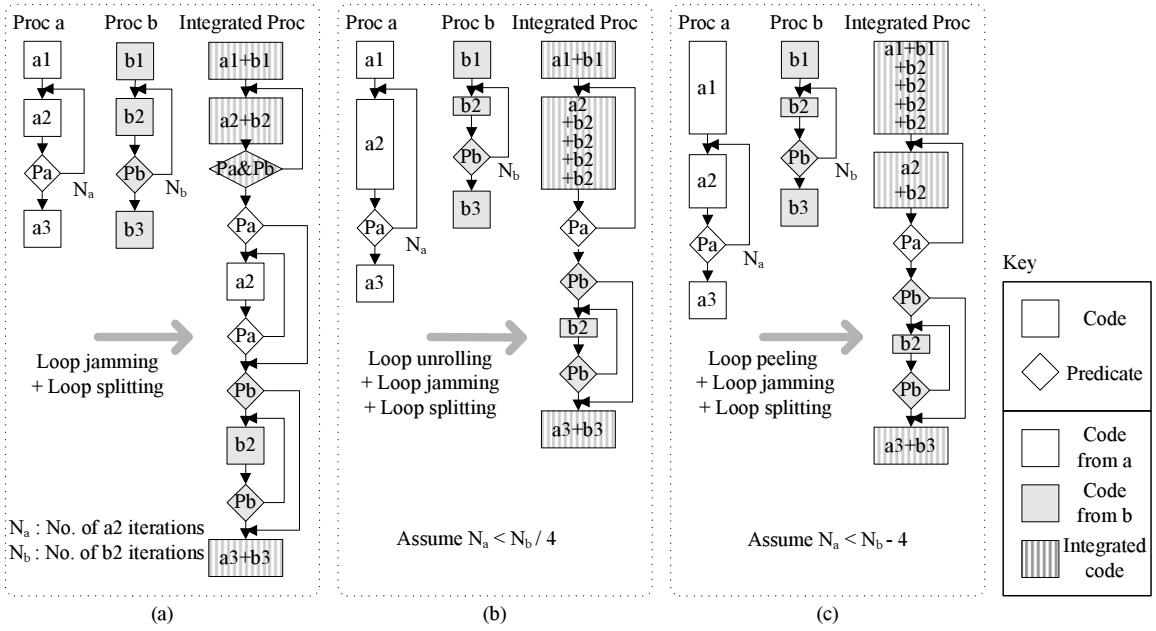


Figure 3.2: STI code transformations for loops

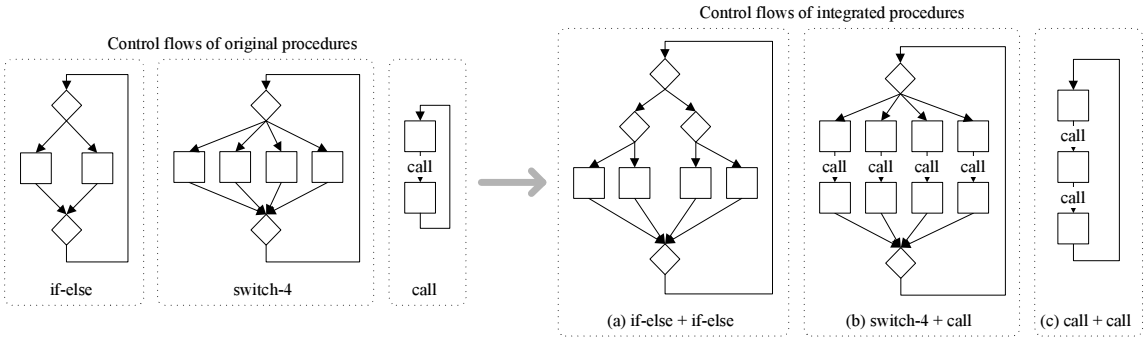


Figure 3.3: STI code transformations for conditionals and calls

loop peeling+jamming+splitting works by peeling one loop then merging peeled operations into code before or after the other loop and jamming the remaining iterations into the other loop body. This transformation is efficient when there are many long-latency instructions before or after a loop body.

Conditionals which can not be predicated and calls which can not be inlined are major obstacles to software pipelining, and hence can limit the performance of applications. STI can be used to improve this code. Figure 3.3 illustrates code transformation examples for the loops with conditionals and calls. It shows control flow graphs of original and integrated procedures before and after STI code transformations for loops with conditionals and calls: (a) if-else + if-else, (b) switch-4 + call, (c) call + call.

Examples in this figure only show control flows of jammed loops for simplicity. When integrating conditionals, all conditionals are duplicated into the other basic blocks. For example, when integrating one if-else with another if-else, both if-else blocks in one procedure are duplicated into both if-else blocks in the other, which results in 4 if-else blocks as shown in Figure 3.3(a). Since resulting basic blocks after integration have both sides of code, the compiler generates a better schedule than when they exist as separate basic blocks. When integrating calls, they are treated like regular statements. Figure 3.3(c) shows the case when integrating a call with another call. Though there is no duplication involved, the resulting code is easier to schedule in that the compiler can find more instructions to fill branch delay slots before calls. Figure 3.3(b) shows the case when integrating conditionals with a call applying the combination of these transformations.

The loop transformations presented above are used based upon the code characteristics which determine software pipelining effectiveness. Table 3.1 presents which transformations to use for a certain combination of code regions A and B based on code characteristics. SWP-Poor, SWP-Fail loops and acyclic code are the best candidates for STI, as these typically have extra execution resources. Integrating an SWP-Good loop with the same type of loop is not generally beneficial because jamming both loops is not likely to improve the schedule of the loop kernel. An SWP-Poor

Table 3.1: STI code transformation matrix

A \ B		Loop			Acyclic
		SWP-Good	SWP-Poor	SWP-Fail	
Loop	SWP-Good	Do not apply STI	STI: Unroll A and jam	Do not apply STI	STI: Loop peeling
	SWP-Poor	STI: Unroll B and jam	STI: Unroll loop with smaller II and jam		
	SWP-Fail	Do not apply STI		STI: Duplicate conditionals and jam	
Acyclic		STI: Loop peeling			STI: Code motion

loop can be used with either SWP-Poor or SWP-Good loops to improve the schedule of the loop kernel by loop jamming. Applying unrolling to SWP-Good loops before loop jamming is useful for providing more instructions to use extra resources in an SWP-Poor loop. Integrating an SWP-Fail loop with either an SWP-Good or SWP-Poor loop should be avoided because jamming those two loop bodies breaks software pipelining of the original loop. An SWP-Fail loop can be integrated with another SWP-Fail loop by duplicating conditionals if any exist. Acyclic code can be integrated with looping (cyclic) code by loop peeling. Lastly, code motion enables integration by moving code in an acyclic region to another acyclic region.

The method how we apply these transformations to the actual code which includes arbitrary control flow and ILP characteristics is described in Chapter 4.

3.3 Experiments

In this section, we examine whether STI complements software pipelining by performing STI to various combinations of looping code.

3.3.1 Target Architecture

Our target architecture is the Texas Instruments TMS320C64x. From TI's high-performance C6000 VLIW DSP family, the C64x is a fixed-point DSP architecture with extremely high performance. It implements VelociTI.2 extensions in addition to

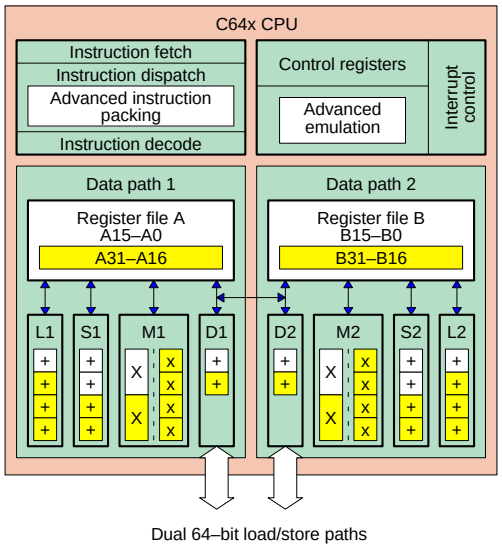


Figure 3.4: Architecture of C64x processor core [100] (Figure courtesy of Texas Instruments)

the basic VelociTi architecture.

The processor core is divided into two clusters which have 4 functional units and 32 registers each. A maximum of 8 instructions can be issued per cycle. Memory, address, and register file cross paths are used for communication between clusters. Most instructions introduce no delay slots, but multiply, load, and branch instructions introduce 1, 4 and 5 delay slots respectively. C64x supports predication with 6 general registers which can be used as predication registers. Figure 3.4 shows the architecture of C64x processor core [99].

C64x DSPs have a dedicated level-one program (L1P) and data (L1D) caches of 16Kbytes each. There are 1024Kbytes of on-chip SRAM which can be configured as a memory space or L2 level cache or both. In our experiments, we use on-chip SRAM as a memory space only. L1P and L2D misses latencies are a maximum 8 cycle and 6 cycles each. Miss latencies are variable due to miss pipelining, which overlaps retrieval of consecutive misses [103].

3.3.2 Compiler and Evaluation Methods

We use the TI C6x C compiler to compile the source code. Original procedures and integrated ones are compiled together with C6x compiler option '-o2 -mt'. The option '-o2' enables all optimizations except interprocedural ones. The option '-mt' helps software pipelining by performing aggressive memory anti-aliasing. It reduces dependence bounds (i.e. RecMII) as small as possible thus maximizing IPC of software pipelined loops. The C6x compiler has various features and is usually quite successful at producing efficient software-pipelined code. It features lifetime-sensitive modulo scheduling [54], which was modified to change resource selection and support multiple assignment code [93], and code size minimization by collapsing prologs and epilogs of software pipelined loops [40].

For performance evaluation we use Texas Instruments' Code Composer Studio (CCS) version 2.20. This program simulates a C64x processor with the memory system listed above and provides a variety of cycle counts for performance evaluation as follows [98].

stall.Xpath measures stalling due to cross-path communication within the processor. This occurs whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle.

stall.mem measures stalling due to memory bank conflicts.

stall.l1p measures stalling due to level 1 program (instruction) cache misses.

stall.l1d measures stalling due to level 1 data cache misses.

exe.cycles is the number of cycles spent executing instructions other than stalls described above.

3.3.3 Overview of Experiments

Figure 3.5 shows an overview of the experiments conducted. Original procedures are integrated constructing different combinations in terms of code characteristics.

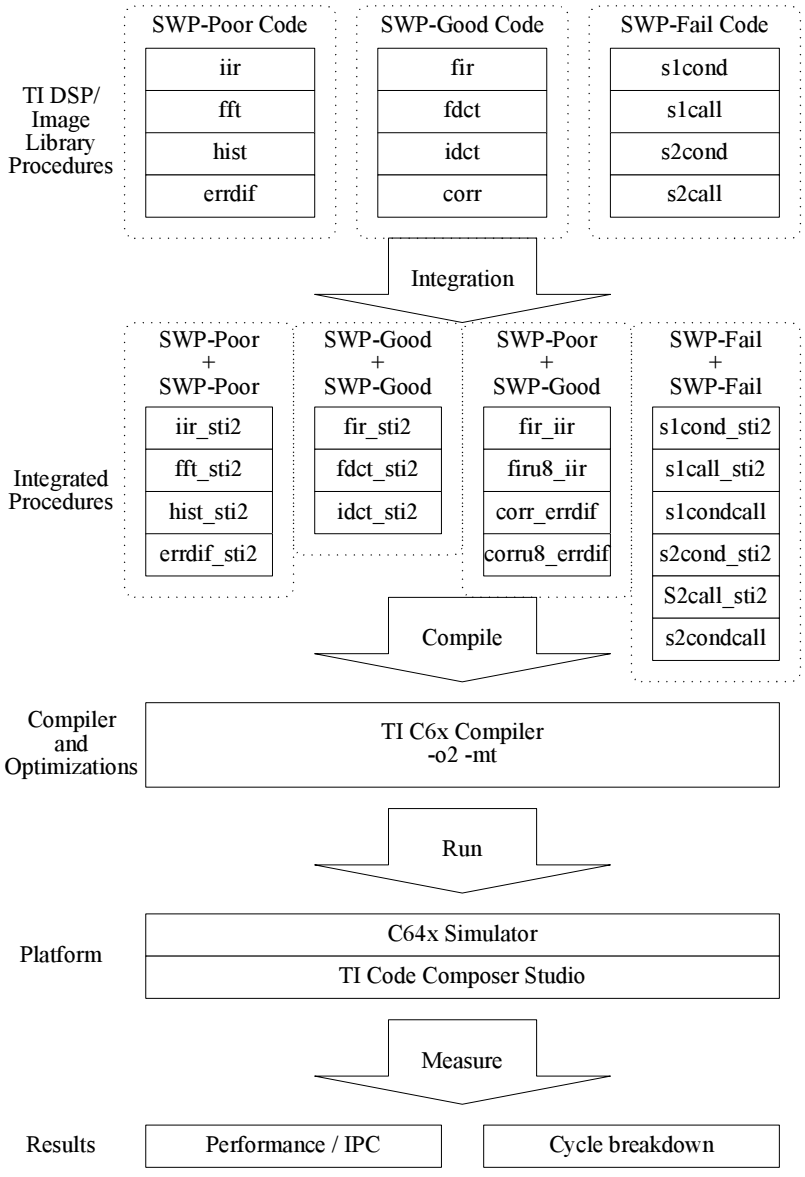


Figure 3.5: Overview of experiments

Each original and integrated procedure is compiled and its performance is measured with TI CCS. Procedures are classified in terms of the characteristics of loops inside. Integrated procedures are written manually in C using code transformation techniques described in Section 3.2.2, constructing different combinations of code. Only combinations of looping code are examined in this work.

For the code where SWP succeeds, we use procedures from the TI DSP/Image Libraries provided with TI CCS. First, we examine integration of SWP-Poor code. Procedures which include dependence bounded loops – *DSP_iir(iir), DSP_fft(fft), IMG_histogram(hist)* and *IMG_errdif_bin(errdif)* – are integrated with themselves using loop jamming. Resulting integrated procedures (with postfix *sti2*) take two different sets of input and output data and work exactly the same as calling the original procedure twice. We assume the parameters, which determine the number of loop iterations, are the same to focus on the effects of transformed code. Therefore, integrated procedures do not include copies of original loops – 'clean-up' loops – but only jammed loops. Having clean-up loops would affect the performance. However, if most iterations are performed by the jammed loops, its influence would be negligible. In order to compare the effects of STI, we also write integrated procedures with SWP-Good loops. Three procedures – *DSP_fir_ gen(fir), IMG_fdct_8x8(fdct)* and *IMG_idct_8x8_12q4(idct)* – are randomly chosen for this purpose.

Second, we examine integration of SWP-Poor and SWP-Good code. We choose combinations of procedures with dependence bounded loops – *iir* and *errdif* and ones with high-IPC resource bounded loops – *fir* and *IMG_corr_gen(corr)*. In addition to basic loop jamming, loop unrolling is used by increasing unroll factors up to 8. The inner loops of *fir* and *corr* are unrolled by 2, 4 and 8 (with postfix *u2*, *u4* and *u8*) then jammed into the inner loops of *iir* and *errdif* respectively. The number of iterations of the inner loops are adjusted so that every iteration runs in the jammed loops hence removing the need for clean-up loops.

For the cases where SWP fails, we build two sets of synthetic benchmarks which characterize the reasons of SWP failures. Synthetic benchmarks represent loops with large conditional blocks and function calls, which cause SWP to fail. The control flow graphs of these procedures appear in Figure 3.3. The first set of benchmarks (with

prefix *s1*) is constructed with the basic unit of mix of simple operations like the inner loop body of *fir*. Since a simple if-else conditional will be predicated by the compiler, a switch-4 (four-way switch) is used for conditional blocks (*s1cond*). For function calls, we insert a modulo operation inside the loop, which leads to a library function call (*s1call*). The second set (with prefix *s2*) is composed of a larger unit block with more instructions from the *fft* loop. An if-else conditional is used (*s2cond*) and a modulo operation is inserted for function calls (*s2call*). For each set of benchmarks, three integrated procedures are written. Two procedures are integrated with themselves (with postfix *sti2*) and one procedure is integrated with the other (*s1condcall* and *s2condcall*). As in previous experiments, the same numbers of loop iterations are assumed.

A simple main function is written for each integrated procedure. They initialize variables and call the two original procedures or the equivalent integrated procedures. In general, input data items are generated randomly but those determining control flows are manipulated so that the control flows take each path equally and alternately. After running programs in CCS, we measure cycle counts spent on original procedures and integrated clones. For each case we perform a sensitivity analysis, varying the number of input items. This changes the balance of looping vs. non-looping code (which includes prolog and epilog code).

3.4 Experimental Results and Analysis

For each integrated procedure, we measure the cycles of the original and integrated procedure as we increase number of input data items. Three sets of data are used for each integrated procedure. Speedups of integrated procedures over original ones are plotted in Figure 3.6. Each line corresponds to speedup of the integrated procedure over original procedures by increasing number of input items. By measuring cycle breakdown as discussed in Section 3.3.2, we divide the whole speedup into five categories: *stall.mem, stall.Xpath, stall.l1d, stall.l1p* and *exe.cycles*. As shown in Figure 3.7, bars above the 0% horizontal line correspond to sources of speedup and

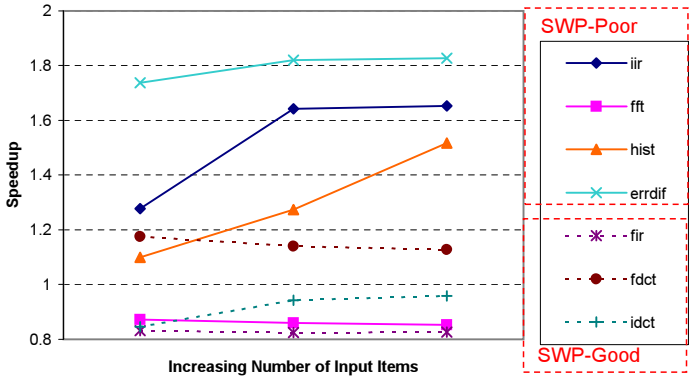
bars below it correspond to sources of slowdown. Code sizes of original and integrated procedures are presented by Figure 3.8.

3.4.1 Improving SWP-Poor Code Performance

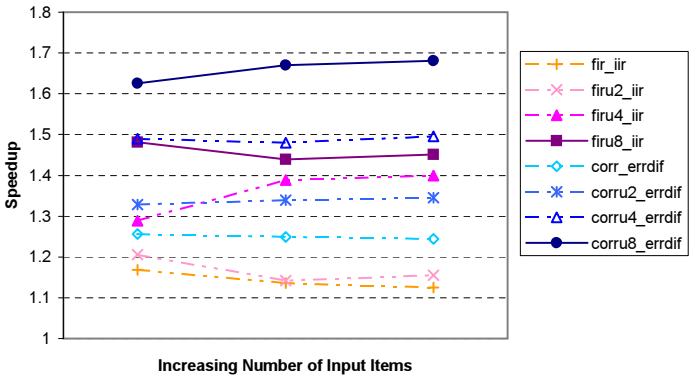
Figure 3.6(a) shows speedups of SWP-Poor code when procedures are integrated with themselves. Speedups of SWP-Good code are also shown with dotted lines for reference. The procedures with SWP-Poor code, which have dependence bounded loops, generally show speedups larger than 1 regardless of number of input items except *fft*. On the other hand, integration is not beneficial for the procedures with SWP-Good code, except for *fdct*.

Figure 3.7(a) identifies sources of speedup and slowdown. Most of the performance improvement comes from *exe.cycles*. These cycles are reduced by improved execution schedules due to integration. In cases except *fft*, schedule cycle counts of software pipelined loops in integrated procedures are improved significantly. Only *fft* does not achieve a speedup from *exe.cycle* because one software pipelined loop in the original procedure is no longer software pipelined after integration. This can happen for loops with a large number of instructions because SWP scheduling algorithm suffers with excessive register pressure and complex dependence between instructions. Stalls other than *stall.Xpath* increase after integration. The increase of *stall.l1p* is expected in that integration forces code size to increase. *stall.l1d* does not increase except for *fft*, where performance is significantly affected. Stalls from memory bank conflicts increase in all cases. We expect that this is caused by the compiler's tendency to align the same types of arrays in the same way. Since accesses to arrays with the same index happen simultaneously in integrated procedures, they cause more memory bank conflicts.

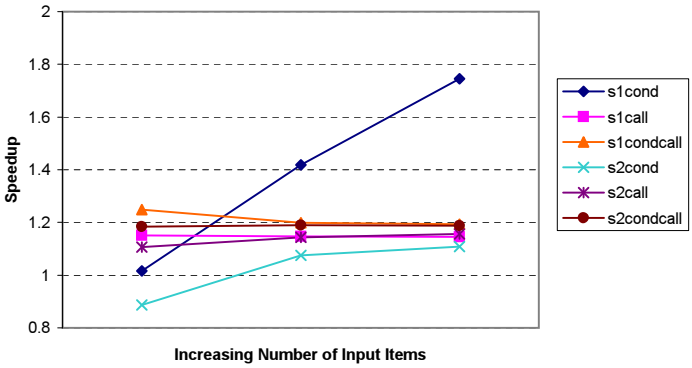
SWP-Poor code is improved by integrating it with SWP-Good code as well as with SWP-Poor code. Figure 3.6(b) shows speedups by integrating *fir* with *iir* and *corr* with *errdif* by applying loop unrolling and loop jamming. Solid lines show the best speedup obtained when unrolling SWP-Good loops by 8 then jamming into SWP-Poor loops for both *fir+iir* and *corr+errdif*. Dashed lines show speedup of non-optimal versions of integrated procedures. Applying unrolling to SWP-Good loops



(a) SWP-Poor + SWP-Poor / SWP-Good + SWP-Good

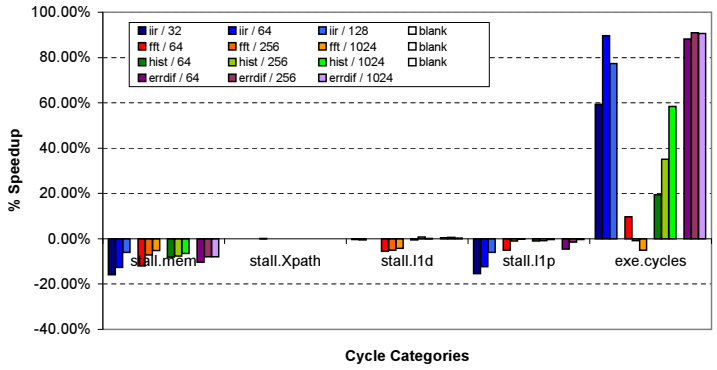


(b) SWP-Poor + SWP-Good

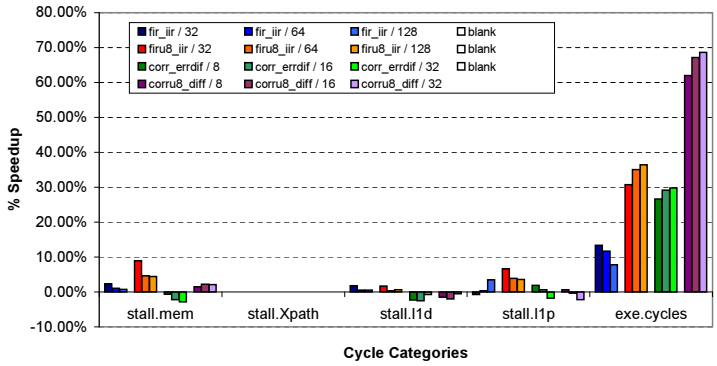


(c) SWP-Fail + SWP-Fail

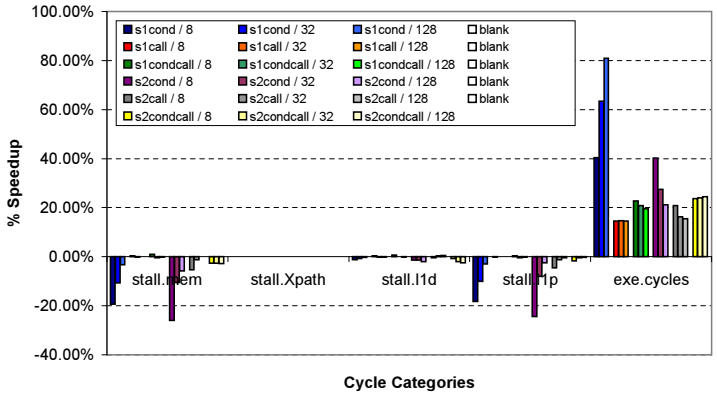
Figure 3.6: Speedup by STI



(a) SWP-Poor + SWP-Poor

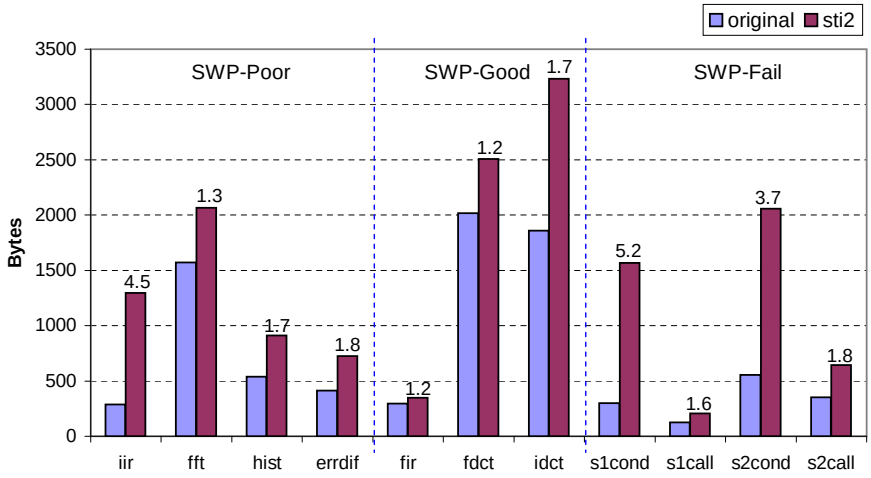


(b) SWP-Poor + SWP-Good

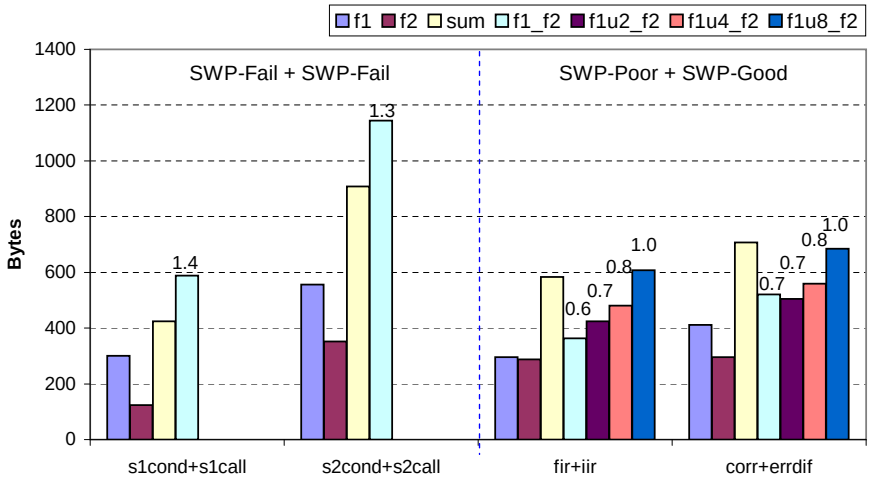


(c) SWP-Fail + SWP-Fail

Figure 3.7: Speedup breakdown



(a) Integration of same procedures



(b) Integration of different procedures

Figure 3.8: Code size changes by STI

before jamming it with SWP-Poor loops significantly improves performance of integrated procedures. Increasing unroll factors consistently increase speedups because instructions from SWP-Good loops fill free empty slots in the schedule of SWP-Poor loops. Figure 3.7(b) verifies that integrated procedures get huge benefits from the improved schedule when applying loop unrolling on top of loop jamming. The impact of stalls is not as consistent as the cases when integrating SWP-Poor code with itself. This is because instructions added by unrolling are not completely independent. Some operations such as memory references can be reused hence reducing the total number of operations. For an example, *stall.mem* decreases after integration contrary to the results in Figure 3.6(a). This is due to the reduced number of total memory accesses by unrolling.

3.4.2 Improving SWP-Fail Code Performance

Figure 3.6(c) shows speedups after integrating SWP-Fail code with SWP-Fail code. All cases but *s2cond* show reasonable speedups over various input items. The cases when integrating a conditional code with the same type – *s1cond* and *s2cond* – show linear speedups by increasing number of input items. It proves that loops suffer non-recurring stalls such as program cache misses. Figure 3.7(c) shows the same pattern as Figure 3.7(a) in that most speedup comes from the improved schedule while stalls are sources of slowdown. However, the positive impact is smaller and the negative impact is bigger hence resulting in smaller speedup numbers. Integrating a conditional with the same type improves the schedule dramatically as well as increasing stalls because duplication increases the sizes of basic blocks significantly.

3.4.3 Impact of Code Size

Code size generally increases after integration and the performance is affected by more program cache misses. Figure 3.8(a) shows code size changes when integrating the procedures with themselves. Each bar shows the code size of the original or integrated procedure. Each number above the bars calculate code size increase ratio. The

procedures which contain conditional codes – *s1cond* and *s2cond* – have significant code size increases because conditional blocks are duplicated into multiple cases as shown in Figure 3.3. *iir* also shows significant code size increase due to the long SWP loop epilog. Other than these, code size increase is less than factor of 2. The absolute code sizes remain smaller than the size of the program cache (16Kbytes). Figure 3.8(b) presents code size changes when integrating different procedures. The first 2 bars show the code sizes of the original procedures (*f1* and *f2*) and the third shows their sum (*sum*). The rest of bars show code sizes of different integrated procedures written by 'loop jamming' (*f1_f2*) and 'loop unrolling + loop jamming' (*f1ux_f2*). Each number above the bars calculate code size increase ratio to code size sum. If there is no conditional, the code sizes of the integrated procedures are smaller than the total code sizes of original procedures as shown in *fir+iir* and *corr+errdif* cases. However, increasing unroll factors causes more code size increase and make it the same as the total code size.

3.5 Conclusions

In this chapter, we present and evaluate methods which allow software thread integration to improve the performance of looping code on a VLIW DSP. We find that using STI complements software pipelining. Loops which benefit little from software pipelining (SWP-Poor) speed up by 26% (harmonic mean, HM). Loops for which software pipelining fails (SWP-Fail) due to conditionals and calls speed up by 16% (HM). Combining SWP-Good and SWP-Poor loops leads to a speedup of 55% (HM). Performance enhancement comes mainly from a more efficient schedule due to greater instruction-level parallelism, while it is limited primarily by memory bank conflicts, and in certain cases by program cache misses.

Chapter 4

Modeling for Efficient STI

4.1 Introduction

STI code transformations for ILP introduced in the previous chapter address how to complement and reinforce software pipelining by selectively applying STI code transformations based upon characteristics of two loops. Each loop's assembly code is examined to characterize the loop based upon speedup from SWP and whether dependence bounds and complex control flow limit SWP's effectiveness. A transformation rule matrix is examined based upon characteristics of the two loops to determine whether to integrate, and if so, how. These methods use a general test which provides a yes/no answer to whether to integrate, and high-level information on which methods to use. However, they do not estimate the performance of the integrated code, limiting the ability to objectively rank potential combinations of loops.

This is important because the number of possible integration scenarios increases significantly as the complexity of threads grows. For example, let's assume the case where Thread 1 and Thread 2 are composed of 4 loops and 2 loops respectively. Depending on which loops are integrated, there are numerous possible integration

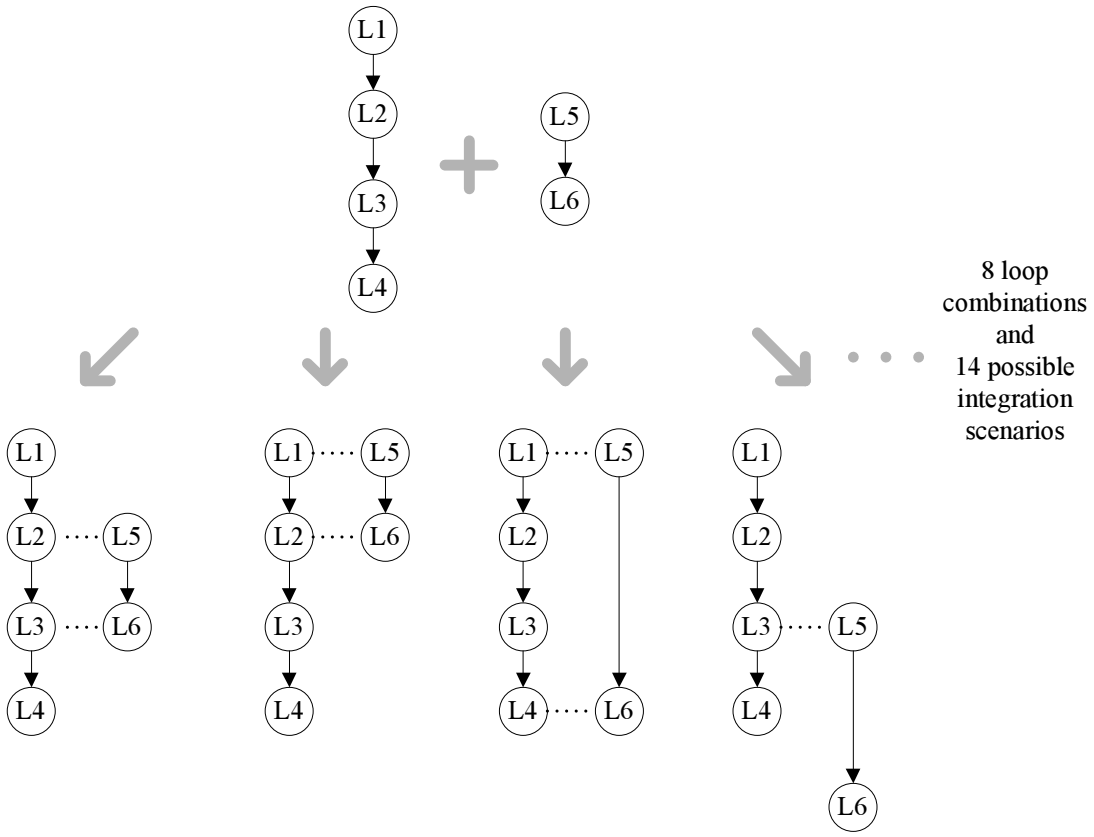


Figure 4.1: Example of multiple integration scenarios

scenarios as shown in Figure 4.1. There are 14 different integration scenarios (8 with one integrated loop, 6 with two integrated loops) in this case. Without any guidance, all 14 integration scenarios should be examined by repeating 14 cycles of integrate/compile/analyze sequences for determining which is the best.

In order to avoid such exhaustive iterations, we incorporate two ideas. The first idea is to quantitatively estimate the performance impact of integration, allowing various integration scenarios to be compared and ranked, leading to more efficient integration. This is a difficult problem because of complexity in both the hardware and the compiler for modern processors (the Texas Instruments TMS320C64x DSP in this case). Software pipelining, predication, branch and load delay slots and limited resources all complicate the prediction of the impact of integration and software pipelin-

ing. We rely on resource modeling, emulating list scheduling and modulo scheduling (via *pseudo-scheduling*) and take into account register pressure and machine features such as predication, and compensate for compiler optimizations as needed.

The second idea is to use these quantitative estimates to guide the alignment of regions in two procedures to yield the fastest code with a small number of iterative compilation cycles. This iterative approach allows inaccuracies in the static performance estimation to be corrected quickly. We demonstrate the methods using code examples from the Texas Instruments DSP and Image/Video Processing Libraries and MiBench. We find that our approach effectively identifies good integration strategies.

This chapter is organized as follows. Section 4.2 describes the methods for analysis, estimation, code transformation and iterative compilation. Section 4.3 presents the hardware and software characteristics of the experiments run, which are analyzed in Section 4.4.

4.2 Methods

Software thread integration (STI) is essentially procedure jamming with intraprocedural code motion transformations which allow arbitrary alignment of instructions or code regions. This alignment allows code to be moved to use available execution resources better and improve the execution schedule. In Chapter 2, we investigated how to select and integrate procedures to enable conversion of coarse-grain parallelism (between procedures) to a fine-grain level (within a single procedure) using *procedure cloning and integration.*

For this work, we focus on an efficient method for finding the best integrated procedure from two independent procedures when there are multiple possible scenarios. The proposed method addresses *the problem of integrating two independent procedures with arbitrary control flow and ILP characteristics.* However, it can be extended to more complicated threads with subroutine call hierarchies and partial data and timing dependencies. These extensions are discussed in Section 4.2.7.

Integration begins with an appropriate representation of original procedures. Analy-

sis of the compiled code provides useful information to identify potentially profitable code combinations. After defining the design space for possible integration scenarios, we develop an efficient searching technique to find the best scenario by incorporating static estimation with iterative compilation.

4.2.1 Program Representation

STI uses the control dependence graph (CDG, a subset of the program dependence graph [31]) to represent the structure of the program: its hierarchical form simplifies analysis and transformation. CDGs are constructed for both host and guest (or secondary and primary) threads to move code from the guest thread to the host thread. Since the STI code transformations have been traditionally done at the assembly language level, the CDGs have been constructed from (compiler-generated) assembly code [25].

For this work, we use transformations at the C source level to avoid duplicating the complexity of a VLIW compiler [87, 88]. For transformation consistency, the CDG is constructed from the source code. In order to do this, we modify some definitions in the original CDG and redefine the graph as the *Simplified CDG (SCDG)*. We limit our scope to structured source code because most C code is structured and most non-structured code can be converted into structured code. The rules to construct the SCDG follow:

1. A list of C statements without control flow change forms a *code* node. A subroutine call is not considered to be control flow change.

2. A loop statement (for, while, do-while) forms a *loop* node and a child node containing statements inside a loop body. The loop type, loop-entry condition and loop incremental expression are stored as attributes of a loop node, if any exist.

3. A conditional statement (if, if-else, switch-case) forms a *predicate* node and child nodes, each of which contains conditionally executed statements. The

```
procA {
  S1;
  if (P1) S2;
  else S3;
  for (L1) {
    S4;
    if (P2) {
      for (L2) S5;
    } /* endif */
    S6;
  } /* endfor */
}
```

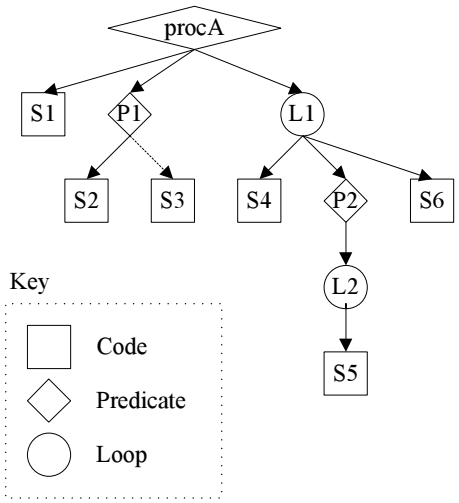


Figure 4.2: Example C source code and corresponding SCDG

conditional type and condition expressions are stored as attributes of a predicate node.

4. The graph is constructed hierarchically by applying these rules repeatedly.

Figure 4.2 shows the example C source code and the corresponding SCDG. The S1,S2,...,S6 labels can represent a list of multiple statements as well as a single statement. The leaf code nodes which contain statements identify the code regions which can be integrated. Though predicate and loop nodes have corresponding C source expressions (P1, P2, L1 and L2), the impact of this code is mostly negligible because it includes considerably less source code than code nodes do, and we therefore do not expand these expressions.

4.2.2 Schedule Analysis

By analyzing the assembly code generated by the target compiler after compilation, we extract useful information for efficient integration. This information is annotated into the SCDG and guides the following steps.

Schedule and profile information

After the original procedures are compiled separately, the assembly code for both procedures is obtained. For analysis, the instructions are divided into basic blocks (BB). We do not treat a subroutine call as a branch instruction for consistency with the previous step. By analyzing assembly code for each basic block, the following information is obtained:

1. Basic schedule information per basic block: number of scheduled cycles (i.e. schedule cycle count) and number of instructions
2. Software pipelining (SWP) information for every inner-most loop: Resource Minimum Initiation Interval (ResMII) and Recurrence Minimum Initiation Interval (RecMII), reason of SWP failure if SWP fails
3. Resource usage per cycle: instruction and its resource usage
4. Register usage per basic block: registers accessed regardless of use and definition

It is possible to construct a control flow graph (CFG) from the assembly code. Based on the CFG and the schedule cycle count of each BB, the static execution time (SET) of each procedure is computed. The SET of sequence of BBs is sum of their schedule cycle counts. If a BB is inside a loop, the SET is the product of its schedule cycle count and loop count. If a BB is conditionally executed, the SET is the product of the schedule cycle count and branch fraction. If it contains a subroutine call, we ignore cycles spent on the callee. Therefore, the SET of the whole procedure is summarized by the following equation:

$$SET_{procA} = \sum_{bb_i \in BB} s_{bb_i} f_{bb_i} n_{bb_i} \tag{4.1}$$

where BB is a set containing all basic blocks of the procedure, bb_i is an element in the set BB and s_{bb_i}, f_{bb_i}, n_{bb_i} represent the schedule cycle count, the execution fraction and the execution count of the bb_i respectively.

Since the execution count (n) and the execution fraction (f) are determined at run time in most cases, the execution profile is necessary to compute the SET. If

the given profile is completely consistent with the actual run, the SET represents the actual run time. A discrepancy between the profile and the actual run may cause the SET and actual run time to differ. Besides the SET, the following steps are affected because they also use the profile to compute profitability. However, it is a common weakness of profile guided optimizations rather than one specific to this method.

If the execution profile is not provided, we can generate it based on static prediction or estimation schemes. For example, we can treat every if-else conditional body equally by setting f to 0.5. Looping code can have higher impact than the other code by fixing n to a certain number, which is considered as priority of loops over acyclic code. The following steps guided by this profile lead to the improvement of the average case instead of the specific case of the provided execution profile.

Modification and annotation of SCDG

In order to use the information from schedule analysis and profiling to guide integration, the information is annotated to the SCDGs of both procedures. Since the SCDG is constructed from the C source code while the information for each basic block is extracted from the assembly code generated by the target compiler, the SCDG must be modified to reflect possible control flow changes caused by the compiler optimizations.

The first source of this change is the compiler's front-end optimizations. The compiler may unroll loops with a fixed number of iterations and a small loop body. If loop unrolling is applied, the SCDG is modified to match with the assembly code. If any other front-end optimizations are performed, the appropriate transformation is followed. The second source is predication. The compiler tries predication for conditionals if there is architectural support. If conditionals are predicated, they no longer cause a control flow change, hence they are treated as one statement during code transformation. This causes the predicate node and its child nodes to be merged into the parent code node or constructed as a separate code node.

Once the control flow of the SCDG and the assembly code are consistent, we annotate each code node in the SCDG with the schedule and profile information given

by the previous step. Matching a code node in the SCDG with a BB in the assembly code is often nontrivial because the assembly code has undergone numerous optimization phases. The compiler optimizations can change control flow as a byproduct of optimization. For example, when a loop is software pipelined, the compiler may generate two separate loops, one SWP loop and one prolog non-SWP loop, if the SWP loop requires minimum number of iterations and the loop count is unknown. Some optimizations may move instructions to near or even further blocks. For instance, loop invariant code motion can move operations inside the loop outside. Global instruction scheduling techniques can move instructions beyond basic block boundaries so that the boundaries between statements in the source code become ambiguous after the code is scheduled. Hence, there is no 'perfect' one-to-one mapping between code nodes in the SCDG and BBs in the assembly code.

The 'best' mapping is pursued by choosing the BB which has the most influence in terms of performance. For a code node under a loop node, only the loop body is considered. When a loop is software pipelined, the SWP kernel is used while the prolog and epilog are ignored. If the prolog non-SWP loop is generated as the result of SWP, it is also ignored. This simplification implies that we care about the schedule which has the most impact on the performance when the loop iterates enough number of times.

For a code node under a predicate node, the BB with the most instructions which correspond to the code node is selected. There are two code generation styles for conditional statements. The compiler prefers a multi-way branch with a jump table when there are multiple conditional bodies more than a certain threshold. This mostly corresponds to a switch-case statement. In this case, each branch target basic block can be matched with a code node under the predicate. Otherwise, the compiler generates cascade branches for the conditional statement such as a simple if-else statement. In that case, the target BBs of the final conditional branches are matched with the code nodes. Since the code order can be reversed after compilation, annotation process should identify the right basic block which corresponds to a code node. Remaining code nodes are matched with the rest of BBs in the assembly code by choosing in-between BBs.

4.2.3 Definition of Alignment Set

There are numerous possible integration scenarios depending on number of code nodes and their control and data dependencies. For efficient exploration, we first define the design space for this problem. Each integration scenario corresponds to a certain *alignment* of code nodes from two procedures, which determines which nodes are integrated with each other and which nodes are left unchanged. A *combination* indicates a pair of code nodes integrated with each other. Therefore, an *alignment* includes at least a single *combination* of code nodes. Let R_A and R_B denote the code node sets for procedure A and B respectively.

$$R_A = \{a_1, a_2, ..., a_m\}, \quad R_B = \{b_1, b_2, ..., b_n\}$$

where $a_1, ..., a_m$ and $b_1, ..., b_n$ represent the code nodes in procedure A and B respectively. Let C_{AB} denote the set including all *combinations*, and L_{AB} denote the set including all *alignments* for procedure A and B.

$$\begin{aligned} C_{AB} &= \{(a_1, b_1), (a_1, b_2), ..., (a_m, b_n)\} = R_A \times R_B \\ L_{AB} &= \{\{(a_1, b_1)\}, \{(a_1, b_1), (a_2, b_2)\}, ..., \{(a_m, b_n)\}\} \\ &= \{x | x \subset C_{AB} \quad \text{and} \quad x \neq \emptyset \quad \text{and} \quad x \text{ is } \textit{legal}.\} \end{aligned}$$

The combination set C_{AB} is the product set of R_A and R_B. Each element in the alignment set L_{AB} is a subset of C_{AB} which is *legal*. Being legal means that two distinct combinations in one alignment do not violate data and control dependencies of the original code. If we conservatively assume that succeeding statements in the original code are always data-dependent on preceding statements and code nodes are supposed to run sequentially in their index orders, it can be rephrased into following two conditions: two distinct combinations in one alignment (1) neither include the same code node (2) nor change the original order of the code nodes. For example, $\{(a_1, b_1), (a_1, b_2)\}$ is not legal because two combinations include the same code a_1. The code a_1 can not be divided into b_1 and b_2 arbitrarily due to possible data dependency and control flow difference between them. The subset $\{(a_1, b_2), (a_2, b_1)\}$ is also not legal because the order of a_1 and a_2 forces b_1 and b_2 to be reversed. While we

conservatively assume the order determines the data dependence, the condition could be relaxed using data flow analysis. Depending on the control dependencies of code nodes in the combinations, the conditions become more complicated. Actual definition of the complete alignment set containing all possible legal alignments is achieved by construction of a decision tree, which represents each alignment as a series of integration decisions, combined with hierarchical SCDG transformation. Details about this process are discussed in Section 5.3.2.

Though STI code transformation enables arbitrary combinations of code nodes, allowing all possible combinations results in explosion of the alignment set size. The number of combinations increases quadratically (i.e. $n(C_{AB}) = n(R_A) \times n(R_B) = mn$) and the upper bound of the number of alignments increases exponentially (i.e. $mn \leq n(L_{AB}) < 2^{mn}$) with the number of code nodes.

In order to reduce the number of alignments, we first include only combinations of code nodes under loop nodes (i.e. cyclic code) in the alignment set, as this code dominates performance. The combinations of other code nodes which are not under loop nodes (i.e. acyclic code) are examined after finding the best alignment of cyclic code. Second, we exclude combinations which have an extremely low probability of speedup such as SWP and non-SWP loop combinations.

4.2.4 Static Profitability Estimation

The alignment set given by the previous step defines the design space for integration. If the size of the alignment set is small, it is not hard to examine all possible scenarios by compiling all integrated procedures. However, it is quite inefficient if there are numerous possible alignments. Therefore, it is necessary to prioritize alignments based on an integration profitability measure.

Static profitability estimation is a modeling method for estimating profitability of a certain alignment. Since we are interested in improving run-time performance by integration, the appropriate measure of profitability is execution cycle count reduction. For modeling, we assume the following: (1) Only the parts of code which have changed after integration (i.e. integrated code nodes) influence the execution cycle

count. (2) Overall profitability of a certain alignment is sum of profitability of code combinations included in the alignment. If the alignment $l_i \in L_{AB}$ includes multiple combinations $c_1, ..., c_n$ (i.e. $l_i = \{c_1, ..., c_n\}$), the *estimated profitability (EP)* for this alignment is given by sum of the profitability of each combination:

$$EP_i = \sum_{c_k \in l_i} p_{c_k} \tag{4.2}$$

The profitability of a certain code combination (p_{c_k}) depends on two factors: (1) The schedule cycle count reduction by integration of code nodes (Δs_{c_k}) and (2) the impact of the integrated code on execution cycle count. The first is determined by how the compiler schedules the integrated code and the second is the product of the execution fraction(f_{c_k}) and the execution count (n_{c_k}). If $a_i \in R_A, b_j \in R_B$ denote code nodes to be integrated and $c_k = (a_i, b_j) \in C_{AB}$ denote the combination of those, the estimated profitability of the combination c_k (p_{c_k}) is computed by the following equations:

$$f_{c_k} = f_{a_i} f_{b_j} \tag{4.3}$$

$$n_{c_k} = min(n_{a_i}, n_{b_j}) \tag{4.4}$$

$$\Delta s_{c_k} = s_{a_i} + s_{b_j} - s_{c_k} \tag{4.5}$$

$$p_{c_k} = \Delta s_{c_k} f_{c_k} n_{c_k} \tag{4.6}$$

where f_{a_i}, f_{b_j}, f_{c_k} are the execution fractions, n_{a_i}, n_{b_j}, n_{c_k} are the execution counts, and s_{a_i}, s_{b_j}, s_{c_k} are the schedule cycle counts of a_i, b_j, c_k respectively.

Pseudo-scheduling

Since the schedule (s_{a_i}, s_{b_j}) and profile ($f_{a_i}, f_{b_j}, n_{a_i}, n_{b_j}$) information are already provided, the only unknown variable is the schedule cycle count of the combination (s_{c_k}). The schedule of integrated code depends on various characteristics of two code nodes such as instruction and cycle counts, used registers and their usage patterns, data dependencies between instructions and so on. Since it is almost impossible to predict the accurate schedule without performing scheduling, we propose a method

called *pseudo-scheduling* to estimate the schedule cycle count of the integrated code. Pseudo-scheduling emulates list scheduling and modulo scheduling for fast estimation. A similar approach which uses dependence and resource bounds is found in [118].

Since VLIW machines use static instruction scheduling, the schedule is embedded in the assembly code. We convert the instruction schedule of each BB into the *resource map (RM)*, a table where number of rows is the schedule cycle count and number of columns is the number of functional units (FU). For each instruction in a BB, there is a mapping between the issued cycle and the issued FU. According to this mapping, each instruction is assigned to a certain entry of the table. Therefore, free spaces in the RM represent idle issue slots in the resulting schedule.

Given resource maps from two code nodes, pseudo-scheduling constructs the schedule of integrated code by moving instructions in one resource map to the idle slots in the other one. Two restrictions are applied during this process. The first one is resource compatibility. In most architectures, instruction and FU types are tied. Depending on its type, an instruction can be issued only to a certain FU or subset of FUs. For a TI C64x DSP, we limit compatibility between different FU types in the same cluster (intra-cluster compatibility) while we allow compatibility between the same FU types in different clusters (inter-cluster compatibility). The second restriction is the data dependencies between instructions. We force instructions to maintain the same cycle distance between them after they are moved to the other resource map. We conservatively apply this restriction even though some instructions are independent. Dataflow analysis would loosen this constraint.

For software pipelined (SWP) loops, the method above does not make sense because instructions from different iterations are overlapped in the SWP kernel. Therefore, we use a specialized pseudo-scheduling method for SWP loops based on modulo scheduling algorithm. In modulo scheduling, the schedule cycle count of a SWP loop kernel (*Initiation Interval (II)*) is constrained by the *Minimum II (MII)*. The MII is the maximum of *Resource MII (ResMII)* and *Recurrence MII (RecMII)*, where the RecMII is determined by the length of recurring dependency and the ResMII is a function of the number of instructions which uses each resource. For ResMII, we use a *Resource Vector (RV)*, where each element is the number of instructions which uses

a specific resource. The number of elements in a RV and the function for computing ResMII (*ComputeResMII*) are architecture dependent. For a TI C64x DSP, there are 20 types of resources involved in ResMII [104].

Besides ResMII and RecMII, the final II depends on the complexity of instruction sequence because the II is determined after the compiler finds a valid SWP schedule. While II is equal to MII when the compiler find a schedule on the first try, II becomes larger than MII as it experiences more trouble to find one. The difference between II and MII (i.e. II - MII) grows with the complexity of the instruction sequence, which generally rises with more instructions, more dependencies between instructions, and more register pressure.

For SWP loops, we estimate the cycle count of the integrated code by predicting RV, ResMII, RecMII, and II. ResMII is computed from the vector summation of two RVs while RecMII is the maximum of two RecMIIs. The final II is estimated by the sum of MII and (II - MII) of the original loops to reflect the schedule complexity in the original code. The following equations summarize the pseudo-scheduling method for software pipelined loops to estimate the schedule cycle count of the integrated code (s_{c_k}).

$$
\begin{aligned}
&ResMII_{c_k} = ComputeResMII(RV_{c_k}) \quad where \quad RV_{c_k} = RV_{a_i} + RV_{b_j} \\
&RecMII_{c_k} = Max(RecMII_{a_i}, RecMII_{b_j}) \\
&MII_{c_k} = Max(ResMII_{c_k}, RecMII_{c_k}) \\
&s_{c_k} = II_{c_k} \quad = MII_{c_k} + (II_{a_i} - MII_{a_i}) + (II_{b_j} - MII_{b_j})
\end{aligned}
$$

For cyclic code combinations, loop unrolling can be applied prior to loop jamming, what we call 'loop unrolling + jamming' transformation. This is powerful in the cases where two loops are asymmetric in terms of loop counts and instruction schedule as discussed in Section 3.2.2. Loop unrolling is applied to the loop which has a larger iteration count because unrolling a loop with a smaller iteration count would decrease the execution count of the integrated loop. For profitability estimation, Equation (4.4)

and (4.5) are slightly modified as follows:

$$n_{c_k} = min(\lfloor n_{a_i}/u \rfloor, n_{b_j}) \quad (4.7)$$

$$\Delta s_{c_k} = u \times s_{a_i} + s_{b_j} - s_{c_k} \quad (4.8)$$

where u denotes the unroll factor, assuming unrolling is applied to a_i.

The schedule of the integrated code after 'loop unrolling + jamming' is also estimated by pseudo-scheduling assuming that the schedule of unrolled code is an identical replicate. The cycle and instruction counts are doubled and the identical resource map (RM) is duplicated after the original one. For a software pipelined loop, the resource vector (RV) and recurrence MII (ResMII) are doubled. After the schedule of the unrolled code is predicted, pseudo-scheduling estimates the schedule of the integrated code in the same way. Multiple unroll factors are examined by doubling the unroll factor up to $u = \lfloor n_{a_i}/n_{b_j} \rfloor \times 2$ when $n_{a_i} > n_{b_j}$. (Otherwise, n_{a_i}, n_{b_j} are interchanged.) Only 2^n unroll factors are allowed because unrolling transformation adds an integer division operator, which introduces a subroutine call unless an operand is 2^n as shown in Figure 4.3(b). After repeating estimation for multiple unroll factors, the best unroll factor is chosen for each loop combination, which results in the maximum estimated profitability.

Register pressure estimation

Since pseudo-scheduling moves instructions from one resource map to the other without considering additional register pressure caused by integration, it ignores the possibility that the compiler may generate different or more instructions to resolve the register conflict. Hence the schedule obtained by pseudo-scheduling is usually better than the real schedule. If many additional spills occur due to heavy register pressure, the schedule of integrated code is much worse than that of original code. Therefore, it is desirable to avoid code combinations which cause spills. We use a method called *register pressure estimation*. We compute number of registers accessed by code nodes and estimate the register pressure of integrated code by adding them. If the sum is larger than the total number of physical registers, we discard it from the combination

set. If there are predicated instructions, the predicate register pressure is estimated in the same way. Following equations summarize register pressure estimation.

$$PReg_{c_k} = PReg_{a_i} + PReg_{b_j}$$
$$GReg_{c_k} = GReg_{a_i} + GReg_{b_j}$$
$$\Delta s_{c_k} = -\infty \quad if \quad ((PReg_{c_k} > MaxPReg) or (GReg_{c_k} > MaxGReg))$$

where $PReg$, $GReg$ are numbers of used predicate and general registers, and $MaxPReg$, $MaxGReg$ are numbers of physical registers.

There are still some cases where differences between the pseudo-scheduling result and the real schedule are large. If the compiler generates different sequences of instructions for the original and integrated code, pseudo-scheduling can not be accurate because it assumes that the same sequence of instructions are generated for the same code after integration. For example, if the compiler succeeds in SIMD optimization for the integrated code though it fails for original code, the generated instructions show a big difference. However, it is extremely hard to predict how every optimization impacts code generation. In reality, instruction scheduling and register allocation are closely coupled, though we use separate models for those. These facts force estimation inaccuracy.

Besides limits of pseudo-scheduling, there is the second source of inaccuracy implied in the profitability estimation model. When we compute the profitability of the alignment, we add the profitability of all combinations included in the alignment as shown in Equation (4.2). Though this simplification is good enough to estimate the integration impact, it does not include every aspect of actual generated code from integrated code.

First, integration introduces the side effect code which causes the performance overhead. For instance, if we perform integration on looping code, the copies of the original loops ('clean-up' loops) also are created with the integrated loop. In actual execution, some additional cycles would be spent on getting in and out of these clean-up loops. Second, the schedule of the non-integrated code could change after integration though it is rare. This could happen due to the increased register pressure caused by interleaving code nodes from two threads. If the original schedule

heavily uses available registers, the schedule could become worse after integration due to reduced available registers by the interleaved code.

Though there are numerous options to improve the accuracy of estimation, we maintain this level of complexity and depend on the target compiler's result to determine the actual profitability. There is a design trade-off between estimation and compilation. If estimation is as accurate as compilation, the best alignment is chosen based on this estimation without iterative compilation. However, the high computation complexity raises time spent on estimation significantly. For this work, the balance between moderate estimation complexity and target compiler's feedback is pursued because we try to define a more general method than one tightly coupled with a certain architecture and compiler.

4.2.5 Code Transformation

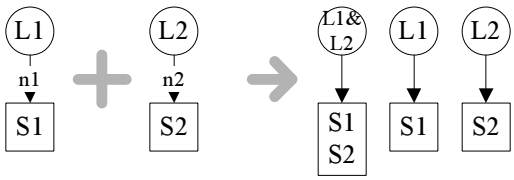
Code transformation is the step where an alignment is converted to the corresponding integrated procedure. Since the alignment determines which parts of the code are integrated, different transformation techniques are applied depending on control dependencies of integrated code nodes. Figure 4.3 illustrates basic STI code transformation techniques and corresponding estimated profitability (EP) depending on the parent node types.

If the parents of both code nodes are loop nodes, they are integrated by loop jamming or loop unrolling + jamming, and the original loops are duplicated to handle possible remaining loop iterations (Figure 4.3(a) and 4.3(b)). When a code node is located under a predicate node, another code node is integrated into every child node (Figure 4.3(c)). If both parents are predicates, the integrated code nodes are created based on all possible paths determined by both predicate conditions (Figure 4.3(d)).

In addition to these basic transformations, there are some advanced STI code transformation techniques to deal with specific situations. Figure 4.4 illustrates examples of advanced STI code transformation techniques. When the integrated code nodes are located in different loop nesting levels, inner/outer loop jamming is applied for STI as shown in Figure 4.4(a). Since only inner-most loops are software

```
Code 1:
  for(i=0;i<n1;i++) S1(i);
Code 2:
  for(j=0;j<n2;j++) S2(j);

Integrated code:
  for(i=0,j=0;i<n1&&j<n2;i++,j++)
    {S1(i); S2(j);}
  for(;i<n1;i++) S1(i);
  for(;j<n2;j++) S2(j);
```

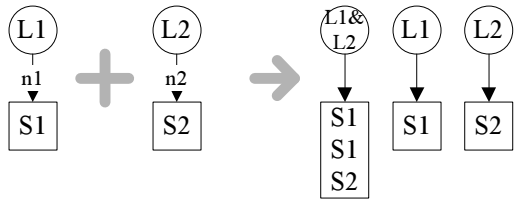
$$EP = \Delta s_{(\mathrm{S1,S2})} \times min(\mathrm{n1}, \mathrm{n2})$$



(a) loop + loop: jamming

```
Code 1:
  for(i=0;i<n1;i++) S1(i);
Code 2:
  for(j=0;j<n2;j++) S2(j);

Integrated code:
  for(i=0,j=0;i<(n1/2*2)&&j<n2;i+=2,j++)
    {S1(i); S1(i+1); S2(j);}
  for(;i<n1;i++) S1(i);
  for(;j<n2;j++) S2(j);
```

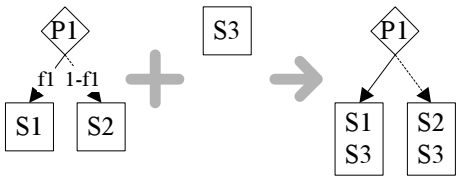
$$EP = \Delta s_{(\mathrm{S1\times 2,S2})} \times min(\mathrm{n1}/2, \mathrm{n2})$$



(b) loop + loop: unrolling + jamming

```
Code 1:
  if (p1) S1;
  else S2;
Code 2:
  S3;

Integrated code:
  if (p1) {S1; S3;}
  else {S2; S3;}
```

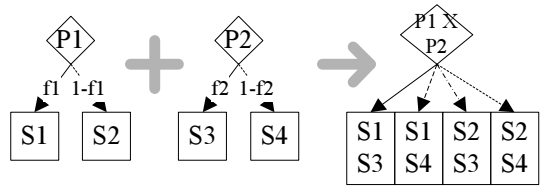
$$EP = \Delta s_{(\mathrm{S1,S3})} \times \mathrm{f1} + \Delta s_{(\mathrm{S2,S3})} \times (\mathrm{1\text{-}f1})$$



(c) predicate + code

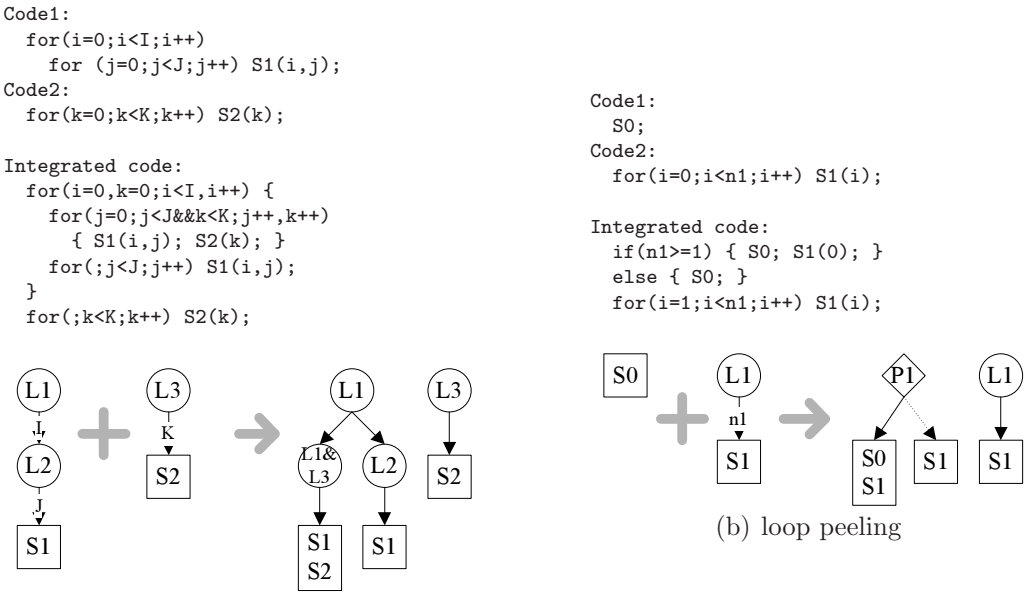
```
Code 1:
  if (p1) S1;
  else S2;
Code 2:
  if (p2) S3;
  else S4;

Integrated code:
  switch ( ((p1!=0)<<1) | (p2!=0)) {
    case 3: S1; S3; break;
    case 2: S1; S4; break;
    case 1: S2; S3; break;
    default: S2; S4;
  }
```

$$EP = \Delta s_{(\mathrm{S1,S3})} \times \mathrm{f1} \times \mathrm{f2} + \Delta s_{(\mathrm{S1,S4})} \times \mathrm{f1} \times (\mathrm{1\text{-}f2})$$
$$+\Delta s_{(\mathrm{S2,S3})} \times (\mathrm{1\text{-}f1}) \times \mathrm{f2} + \Delta s_{(\mathrm{S2,S4})} \times (\mathrm{1\text{-}f1}) \times (\mathrm{1\text{-}f2})$$



(d) predicate + predicate

Figure 4.3: Basic STI code transformation techniques



(a) inner/outer loop jamming

(b) loop peeling

Figure 4.4: Advanced STI code transformation techniques

pipelined, it is often required to integrate SWP loops with different loop nesting levels. For integration of acyclic code and cyclic code, loop peeling is applied as shown in Figure 4.4(b). By combining and hierarchically applying these transformations, any alignments can be successfully realized. Implementation of hierarchical SCDG transformation is described in Section 5.3.2.

The transformation techniques presented above are targeted for the general case. Depending on the situation, some code can be removed if unnecessary. For example, if both loop counts are fixed and the same, copies of the original loops (clean-up loops) in Figure 4.3(a) are not necessary. Transformation leaves opportunities for further optimizations such as induction variable elimination and loop invariant code motion. The jammed loops may have two induction variables unnecessarily after integration, which can be reduced into one. For instance, either i or j in Figure 4.3(a) and 4.3(b) can be eliminated if neither S1 nor S2 defines it. If there is the similar code which operates on i and j separately, it can be reduced as well. After induction variable elimination, loop invariant code motion can be applied additionally. Figure 4.5 shows an example of these optimizations. Since elimination of instructions or

Base integrated code:

```
for(i=0,j=0;i<n1&&j<n2;i++,j++) {S1(i); S2(j);}
for(;i<n1;i++) S1(i);
for(;j<n2;j++) S2(j);
```

After induction variable elimination:

```
for(i=0;i<min(n1,n2);i++) {S1(i); S2(i);}
for(;i<n1;i++) S1(i);
for(j=i;j<n2;j++) S2(j);
```

After loop invariant code motion:

```
n3 = min(n1,n2);
for(i=0;i<n3;i++) {S1(i); S2(i);}
for(;i<n1;i++) S1(i);
for(j=i;j<n2;j++) S2(j);
```

Figure 4.5: Additional optimizations to base integrated code

moving instructions out of the loop directly improves the schedule of the loop, its impact is significant especially for small loops.

4.2.6 Iterative Compilation

By preceding steps, we define the set of alignments and estimate the profitability of each alignment by the static profitability estimation. Due to necessary simplifications implied in the estimation model, we use an *iterative compilation* approach to verify the integrated procedure selected by estimation and correct possible inaccuracy. Iterative compilation is often used in compiler optimizations such as finding effective compilation sequences [9], and loop transformations [63]. The method for iterative compilation presented here is a heuristic approach to find the best alignment with the fewest compilation cycles (i.e. cost). We define the following distinct measures for an alignment:

1. Estimated profitability of the alignment l_i (EP_i) is obtained by Equation (4.2) and (4.6). Initial s_{c_k} is computed by pseudo-scheduling but is updated by the

real schedule cycle count obtained after each compilation cycle. Therefore EP_i converges to XP_i.

2. Expected profitability of the alignment l_i (XP_i) is computed by the same way as EP_i but uses s_{c_k} given by the real schedule of the integrated code after compilation.

3. Actual profitability of the alignment l_i (AP_i) is the static execution time (SET) difference between original procedures and the integrated procedure. The SETs are obtained by analyzing its assembly code as described in Section 4.2.2.

The EP_i is the only measure which can be computed initially before any compilation. The XP_i reflects the schedule of the integrated code but still implies some inaccuracy from side effects of code transformation. The AP_i reflects the real schedule of the whole integrated procedure. The inequality $EP_i \geq XP_i \geq AP_i$ therefore holds for most cases because the schedule from pseudo-scheduling is more optimistic than the real schedule and side effects of code transformation usually have a negative impact. The actual profitability of the integrated procedure is determined by AP_i while we use EP_i for prioritizing alignments and XP_i for determining when the iterative compilation cycle stops.

Figure 4.6 shows the algorithm for iterative compilation. First, the alignment (l_i) with the largest EP is chosen and an integrated procedure is generated (Line 1-4). After compilation, the schedule of integrated code is determined from the assembly code and the XP and AP are computed (Line 5). Since the real schedules of the code combinations (i.e. integrated code) included in the alignment are obtained, the EPs of l_i and other alignments which share the same combinations are updated (Line 6). After this update, the alignments with a negative EP are removed from the alignment set because compilation may have identified unprofitable code combinations (Line 7). Then we choose the alignment with the maximum of updated EPs again (Line 8). If the alignment is the same as the one just compiled, the iterative cycle stops (Line 9-10). Otherwise, another compilation cycle is performed with the newly chosen alignment (Line 12). Once the cycle stops, we choose the best integrated procedure

1: Choose $l_i \in L_{AB}$ where $EP_i = Max(EP_1, ..., EP_n)$;
2: **loop**
3: Perform code transformation for l_i;
4: Compile integrated procedure;
5: Compute XP_i and AP_i;
6: Update EP_k for all $l_k \in L_{AB}$ where $l_i \cap l_k \neq \emptyset$;
7: Remove all l_k where $EP_k < 0$ for all $l_k \in L_{AB}$;
8: Choose $l_j \in L_{AB}$ where $EP_j = Max(EP_1, ..., EP_{n'})$;
9: **if** $j = i$ **then**
10: **break;**
11: **end if**
12: $i \leftarrow j$;
13: **end loop**
14: Choose $l_i \in L_{AB}$ where $AP_i = Max(AP_1, ..., AP_{n'})$;

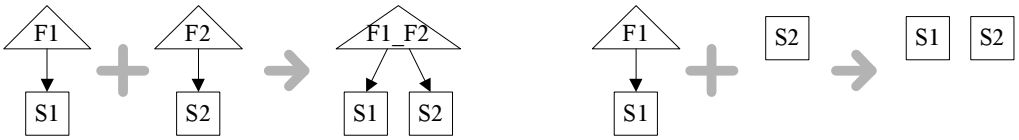
Figure 4.6: Algorithm for iterative compilation

from compiled ones based on the AP (Line 14).

This algorithm has two important implications. The first is a *feedback mechanism* implemented by an EP update. Since this feedback reprioritizes alignments based on updated EPs, it minimizes the impact of pseudo-scheduling inaccuracy on future alignments by considering previous compilation results. The second is a *decision mechanism* implemented by a comparison of EP and XP. After an EP update, EP is the same as XP for alignments which have been compiled. By comparing the EP(=XP) of compiled alignments with the EP of yet-to-be-compiled alignments, the algorithm effectively filters out the alignments which are not worthwhile because the EP generally determines the upper bound of the XP.

4.2.7 Possible Extensions

Though the proposed method addresses integration of two *completely independent* procedures *ignoring subroutine calls*, it can be extended to following two directions.



(a) procedure cloning (b) procedure inlining

Figure 4.7: STI code transformations for subroutine calls

Subroutine call extension

Analysis and transformation can be extended to more complex threads, which include subroutine call hierarchies. When the SCDGs for original threads are constructed, a subroutine call node is created for each call. The SCDG of the subroutine is constructed below the subroutine call node. Then, alignment set definition and static profitability estimation are performed as before, but including code nodes from subroutines.

STI code transformations must be extended to deal with subroutine calls. Depending on the location of the code node to be integrated (i.e. alignment), subroutine calls must be transformed as well. If no code node under subroutine call nodes is integrated, no change is required for integration. If both integrated code nodes come from subroutine calls, those subroutines are integrated creating a procedure clone. If a code node coming from a subroutine is integrated with a code node not coming from any subroutines, the subroutine is inlined so that integration is done in the same level of call hierarchy. Figure 4.7 illustrates those transformations, where triangle nodes denote subroutine call nodes in the SCDGs. Procedure cloning and inlining must be carefully applied because they could cause a significant code size increase. (Procedure inlining under a code expansion constraint for embedded systems is discussed in [70].) This extension makes STI for ILP applicable for threads with arbitrary sizes, which may include multiple levels of subroutine call hierarchies.

Dependent thread extension

STI for ILP can be extended to handle threads with data dependencies and timing constraints. Data flow analysis can identify the data dependencies between code nodes and this information can be used during alignment set definition. As discussed in Section 4.2.4, data independence between the code nodes in one thread (intra-thread data independence) relaxes the condition for being a 'legal' alignment. On the other hand, data dependence between the code nodes across two threads (inter-thread data dependence) adds more constraints for being a 'legal' alignment. While the alignment set is constructed, this data dependency information is used to filter out alignments which violate data dependency constraints. For instance, if there is a data dependency between a_i and b_j, every alignment which reverses the execution order of a_i and b_j would be dropped from the alignment set.

Timing constraints between two threads can be treated in a similar way assuming that the execution behavior of the code is predictable in terms of timing. If a timing constraint forces a certain execution order of the code nodes, it can be treated as a data dependence. If the specific code nodes are required to be executed simultaneously, the code combination containing two code nodes is always included in the alignment set (positive constraint). Synchronization primitives between two threads can be implemented similarly. For example, the code waiting on a semaphore must be preceded by the code signaling it. The alignments which violate this execution order will be removed from the alignment set. The code including critical sections between two threads, implemented by a mutex, will not run simultaneously. Hence the code combinations including those are excluded from the combination set.

Since the alignment set size decreases as more constraints are added, the probability of having beneficial alignments reduces as well. However, this extension is powerful because it expands applicability of STI for ILP to partially dependent threads.

4.3 Experiments

4.3.1 Target Architecture

Our target architecture is the Texas Instruments TMS320C64x. From TI's high-performance C6000 VLIW DSP family, the C64x is a fixed-point DSP architecture with extremely high performance. The processor core is divided into two clusters with 4 functional units and 32 registers each. A maximum of 8 instructions can be issued per cycle. Memory, address, and register file cross paths are used for communication between clusters. Most instructions introduce no delay slots, but multiply, load, and branch instructions introduce 1, 4 and 5 delay slots respectively. 6 general registers can be used as predication registers.

4.3.2 Compiler and Evaluation Methods

We use the TI C6x C compiler to compile the source code. Original procedures and integrated ones are compiled together with C6x compiler option '-o2 -mt'. The option '-o2' enables all optimizations except interprocedural ones. The option '-mt' helps software pipelining by performing aggressive memory anti-aliasing. It minimizes dependence bounds (i.e. RecMII), maximizing utilization for software pipelined loops. The C6x compiler has various features and is usually quite successful at producing efficient software pipelined code. For performance evaluation, we use the stand-alone simulator in Texas Instruments' Code Composer Studio (CCS) version 2.20. By enabling profiling, the execution cycles of original and integrated procedures are measured.

4.3.3 Procedure Selection and Integration

Four pairs of procedures are chosen from TI DSP/Image Libraries [101, 102] and MiBench benchmark suite [42] for integration. Among procedures which are relevant to main program workloads, various procedures with multiple loops (SWP or non-

Table 4.1: Summary of procedure pairs used for experiments

Case	Proc.A	Name	Proc.B	Name	Program	Benchmark suite
(a)	DB	DSP_bexp	IH	IMG_histogram	N/A	TI DSP/Image Lib.
(b)	XF	sha_transform	BR	byte_reverse	sha	Mibench/security
(c)	BCE	BF_cfb64_encrypt	BSK	BF_set_key	blowfish	Mibench/security
(d)	AC	Autocorrelation	CLTP	Calculation_of_the_LTP_parameters	gsm	Mibench/telecomm.

Table 4.2: SCDG characteristics of procedure pairs and STI results

Procedure pairs	Proc.A L	P	R_A	Proc.B L	P	R_B	Integration space $R_A \times R_B$	C_{AB}	L_{AB}	Comp. cycles	Comp. time (sec.): ICA/Exhaustive	Best speedup
(a) DB + IH	2	0	2	2	0	2	4	4	5	2	2.74/6.52	6.55%
(b) XF + BR	5	0	5	1	0	1	5	5	5	5	8.75/8.75	9.40%
(c) BCE + BSK	2	3	4	3	0	3	12	4	4	1	1.02/3.87	10.13%
(d) AC + CLTP	4	3	7	5	2	5	35	7	12	1	5.73/55.73	6.86%

L: Loop nodes, P: Predicate nodes, Comp.: Compilation, ICA: Iterative Compilation Algorithm

SWP), conditionals or calls are selected to show the effectiveness of our methods. Table 4.1 summarizes selected procedures for experiments. For these four pairs of procedures, the iterative compilation method is applied and examined by generating all possible integrated procedures. Although analysis and estimation are automated, code transformation is performed manually for the experiments presented in this chapter. Automatic integration is also tried by our automation tool chain and results appear in Chapter 5.

For the alignment set, we include only looping code as discussed in Section 4.2.3. For integration of looping code, we apply loop jamming or loop jamming + unrolling transformation. General transformation techniques are performed; further optimizations discussed in Section 4.2.5 are performed for loops with fixed iteration counts. For simulation, random test inputs are generated, and variable-iteration loops are adjusted to execute at least 64 times. The profile used for static profitability estimation is assumed to be consistent with the actual run in order for the fair comparison of profitability measures from static estimation and simulation.

4.4 Experimental Results and Analysis

Figure 4.9 shows the SCDGs of original procedure pairs used for experiments annotated with basic schedule and profile information. The profile information annotated on edges represents execution count or execution fraction. Below a code node, the schedule information is annotated with instruction and cycle count (i/c) and predicate and general registers used (p-g). The letter 'S' in a loop node indicates that it is software pipelined and 'call' appears in the annotation if there is a subroutine call inside a code node. Only code nodes under loop nodes are annotated because only combinations of those are considered for integration.

Table 4.2 summarizes the characteristics of original procedures, integration space and result of the iterative algorithm runs. The L, P and R_A (R_B) columns represent number of loop nodes, predicate nodes and code nodes located under loop nodes. The C_{AB} and L_{AB} columns show the number of combinations and alignments for

integration. The actual combination set used for integration is a subset of the whole combination set (i.e. $C_{AB} \subset R_A \times R_B$) because some combinations such as SWP/non-SWP pairs, or ones with heavy register pressure are discarded by the estimation process. The last three columns show the number of compilation cycles performed by the iterative algorithm, the comparison of required compilation time with (ICA) and without our method (Exhaustive), and the speedup of the integrated procedure finally selected by the iterative algorithm. (Compilation time is measured on a Pentium 4 Windows machine.) While only 1 compilation is needed for case (c) and (d), 2 and 5 compilations are performed for case (a) and (b) respectively. Compared with the total number of all possible alignments (L_{AB} column) and compilation time required for compiling all alignments, there are significant compilation savings by our method in terms of both compilation cycles and time. For example, Figure 4.8 shows how the iterative compilation algorithm selects Alignment 2 after 2 compilation cycles for case (a).

Figure 4.10 shows different profitability measures discussed in Section 4.2.6 obtained after generating and compiling all possible integrated procedures. For convenience, the AP is measured by simulation instead of static execution time measurement described in Section 4.2.2 because they are equivalent when the profile is consistent with the actual run. Though the EP is updated after every compilation cycle, this figure depicts the initial EP computed by the static estimation with no feedback. Alignments (i.e. integrated procedures) are numbered by the descending order of their initial EPs from the left. The alignment finally selected by the iterative algorithm (Alignment 2, 3, 1 and 1 for each case) is marked with its % speedup.

The most important observation from this figure is that the procedure selected by our method is the actual best one for every case. The marked procedure shows the highest AP after actual compilation and simulation. In case (c) and (d), the alignment with the largest initial EP (i.e. Alignment 1) is chosen as the best after compilation of itself. Recall that the algorithm decides whether it continues or not based on a comparison of EP and XP. In both cases, the iteration stops after the first compilation because $XP_1 > EP_2$. In case (a), one more compilation cycle is performed because $XP_1 < EP_2$ but $XP_2 > EP_3$. (Note that actual EP values change

Initial state

L	EP	XP	AP	Action
1	528			Compile
2	512			
3	64			
4	16			
5	16			

After compilation of Alignment 1

L	EP	XP	AP	Action
2	512			Compile
1	504	504	96	
3	64			
4	16			
5	-8			Remove

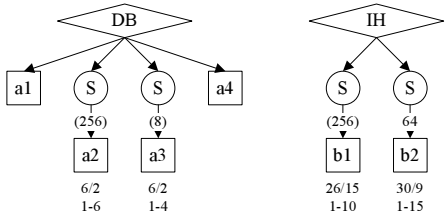
After compilation of Alignment 2

L	EP	XP	AP	Action
2	512	512	132	Stop
1	504	504	96	
3	64			
4	16			
5	Removed			

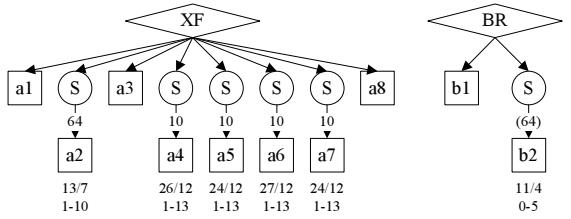
Final State

L	EP	XP	AP	Action
2	512	512	132	Chosen
1	504	504	96	
3	64			
4	16			
5	Removed			

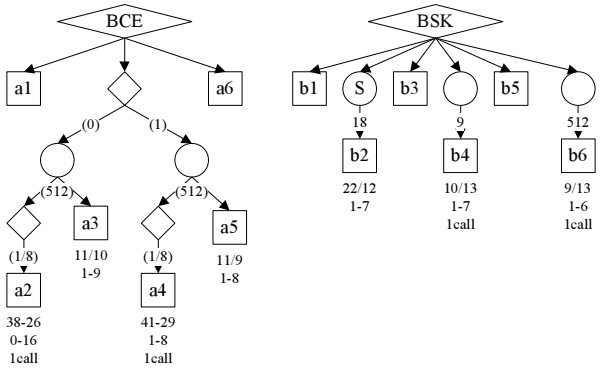
Figure 4.8: Iterative algorithm runs for case (a) DB + IH



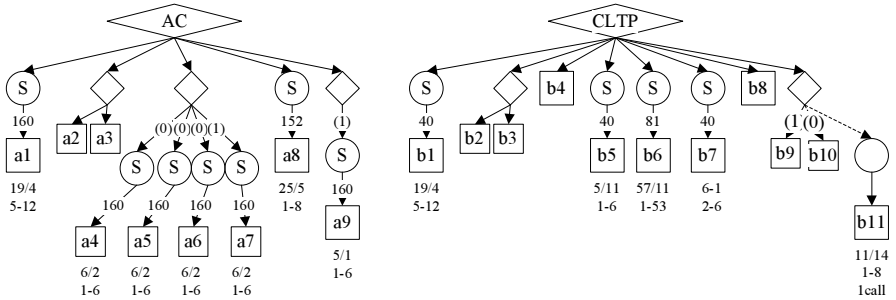
(a) DB + IH



(b) XF + BR

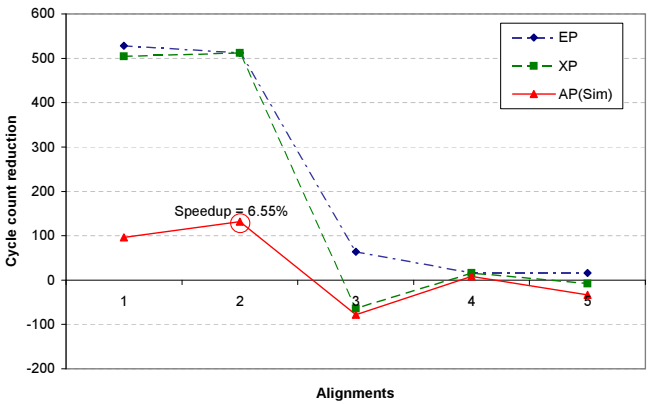


(c) BCE + BSK

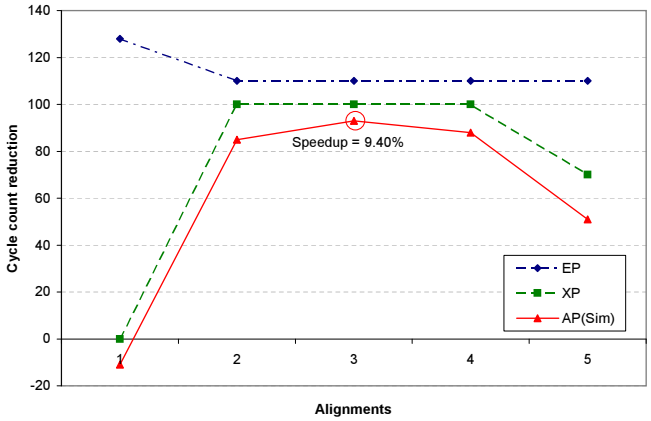


(d) AC + CLTP

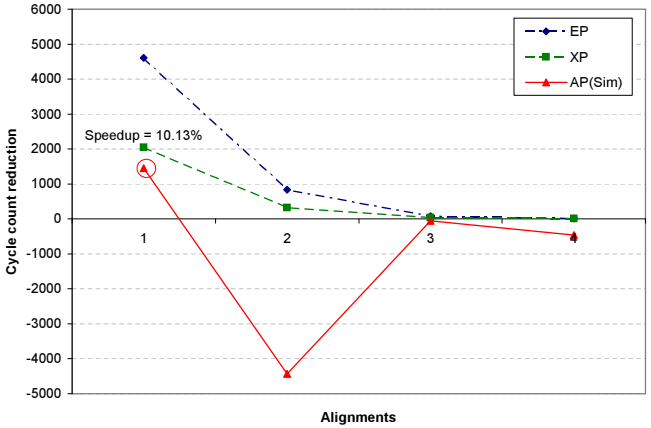
Figure 4.9: SCDGs of original procedure pairs with basic annotations



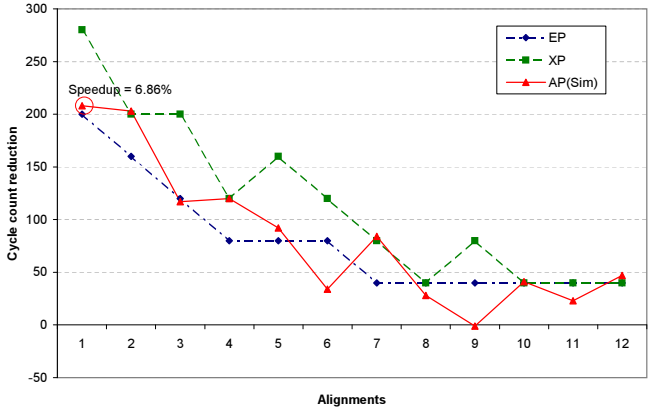
(a) DB + IH



(b) XF + BR



(c) BCE + BSK



(d) AC + CLTP

Figure 4.10: Different profitability measures of all alignments for each integration case

slightly after compilation because EPs are updated as shown in Figure 4.8.)

The case (b) is an example of the worst case scenario because there are 5 alignments and we need to compile all of them to find out which is the best. As shown in Figure 4.9(b), there are 5 distinct combinations of loops, each of which constitutes an alignment. The worst case can happen if there is a large estimation error and alignments cannot take advantage of feedback. The alignment which is estimated as the most beneficial (Alignment 1) turns out to be non-beneficial after the first compilation cycle. This is because the schedule cycle count estimated by pseudo-scheduling differs much from the actual compilation result for this integrated loop. The EPs of remaining alignments (Alignments 2 through 5) are equal and none of them is updated after compilation because there are no common combinations shared among alignments. This means that all remaining alignments are estimated to be equally beneficial. Hence, the best alignment is identified only after compiling all of them.

Regarding relationship between EP, XP and AP, they show similar trends, meaning that static estimation works well for cases except (b). The inequality $EP_i \geq XP_i$ the foundation of decision mechanism, holds for all alignments in case (a), (b) and (c) showing that estimation is optimistic. In case (d), $EP_i \geq XP_i$ does not hold for most alignments. This is due to one of the additional optimizations discussed in Section 4.2.5. In this case, induction variable elimination is applied in order to improve the integrated code because all loops has fixed iteration counts and the original code schedule is too dense to be improved by general code transformation. Though initial EPs are less than XPs, EPs are updated after compilation cycles, quickly converging to XPs. Therefore, the iterative compilation algorithm is not disrupted much with additional optimizations.

Figure 4.11 shows the histogram for schedule cycle count errors between pseudo-scheduling and real scheduling results of all 20 code combinations included in 4 cases. Most errors are negative indicating that pseudo-scheduling is more optimistic than the real schedule. The errors are within 2 cycles for 14 combinations including 6 zero errors from perfectly accurate estimation. Among 6 combinations in which absolute errors are larger than 2 (<-2 bar), two correspond to the SWP loop combinations in case (b). The other 4 combinations come from case (c) where the integrated code

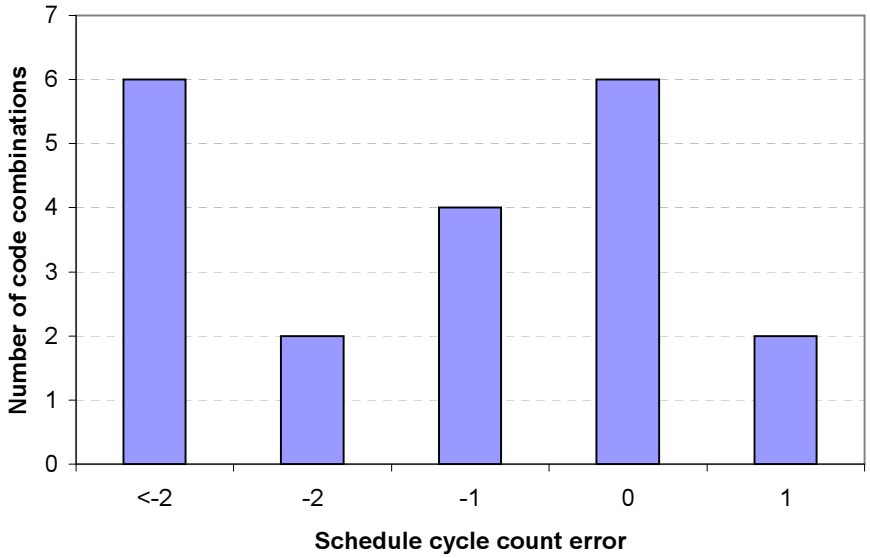


Figure 4.11: Histogram for schedule cycle count errors between pseudo-scheduling and real scheduling results

includes subroutine calls. The errors are relatively large because pseudo-scheduling does not treat a call instruction specially while the real compiler does. However, the errors occur in all combinations so they do not affect the EP and XP relationship. Two combinations with positive errors correspond to SWP loop combinations in case (d) where we apply induction variable elimination.

Experimental results demonstrate that the static profitability estimation via pseudo-scheduling works well despite some exceptions. The iterative compilation algorithm compensates for possible estimation inaccuracy and finds the best integration scenario with reasonable efficiency. When estimation works well, the algorithm effectively filters out the integrated procedures for which compilation is unnecessary by comparing realized profitability (XP) with potential profitability (EP). For the cases where estimation is not good enough, the feedback mechanism in the algorithm enables convergence of estimated profitability (EP) with the real profitability (XP) by applying previous compilation results to the future alignments.

4.5 Conclusions

In this chapter we propose an efficient method for finding the best integration scenario given two independent procedures with arbitrary control flow and ILP characteristics. Static profitability estimation relying on resource modeling, pseudo-scheduling and register pressure estimation allows various integration scenarios to be compared and ranked, identifying more beneficial ones. The iterative compilation algorithm compensates for possible estimation inaccuracy by decision and feedback mechanism, enabling quick selection of the best scenario with the fewest compilations. By experimental runs on the TI C64x DSP with code examples from the TI DSP/Image Libraries and MiBench, we demonstrate that our estimation shows reasonable accuracy for most cases and the iterative algorithm finds the best integrated procedure with limited compilations despite the difficulty of accurate estimation.

Chapter 5

STI for ILP Tool Chain

5.1 Introduction

This chapter describes our tool chain for automatic software thread integration for instruction level parallelism : *STI for ILP Tool Chain*. For STI for ILP, we apply code transformations at the C language level instead of the assembly language level for practical reasons. Assembly level integration requires a register allocator and an instruction scheduler that targets the machine and accurately models timing. It is nontrivial for commercial VLIW DSPs like the TI C6x used in this research.

In order to implement C source level transformation, we leverage the SUIF2 (Stanford University Intermediate Format) compiler, a research compiler developed at Stanford University [1]. In our tool chain, the C source code is converted into the appropriate graph representations. Based on the information from the assembly code generated by the target compiler, the possible STI scenarios are enumerated and prioritized by a profitability analysis. Automatic code transformation is performed on the graph representation of the program, which is later converted into the the C source code.

One of the biggest advantages of this implementation is that the code transfor-

mation functionality in our tool chain is platform-independent. Our tool chain can be ported to other VLIW DSP platforms by modifying the profitability estimation, which now targets the TI C6x DSP.

This chapter is organized as follows. Section 5.2 gives an brief overview of the STI for ILP Tool Chain. Section 5.3 presents operation of our tool chain in detail. In Section 5.4, we show the experimental results from automatically integrated procedures by revisiting the procedures extracted from the TI Image/DSP Libraries and MiBench used in the previous chapter. Section 5.5 discusses some possible future extensions for the current STI for ILP Tool Chain.

5.2 Tool Chain Overview

Figure 5.1 shows the overview of the tool chain and Table 5.1 lists programs included in the tool chain. The C source code for original threads are fed into both the target compiler (*cl6x*) and the SCDG generator (*c2scdg*). The assembly code generated by the target compiler is interpreted by our assembly processing tools (*bbi.pl/bbi2cfg*) then converted into basic block information files and control flow graphs. Then, the SCDGs are modified and annotated with the basic block and profile information by feeding directive files to the tools (*scdg_mod/ann*). By taking the annotated SCDG files from both threads, the main STI program (*scdg_sti*) generates all possible integrated procedures in the form of transformed SCDGs, performing static profitability estimation, alignment set definition, and SCDG transformation. Finally, the generated SCDGs are converted into the C source code by the SCDG-to-C converter (*scdg2c*). While the tools which process the TI C6x assembly code (*bbi.pl/bbicfg*) are platform-dependent, the tools which works on C to SCDG and SCDG to C (*scdg2c*, *scdg_mod/ann*, *scdg_sti* and *c2scdg*) are platform-independent, except for the profitability analysis built in the *scdg_sti* program.

Basic outputs and visualizations are supported by the STI for ILP Tool Chain. Both CFGs from the assembly code and SCDGs from C source code can be generated in the DOT language and visualized by DOT tools [36]. The main STI program

Table 5.1: List of tools in STI for ILP Tool Chain

Programs	Functions
cl6x	TI C6x compiler
bbi.pl	Parse TI C6x assembly and convert into basic block info. (BBI) file
bbi2cfg	Read BBI file then construct CFG
c2scdg	Convert C program into SCDG format via SUIF pass
scdg_mod	Modify SCDG for consistency with assembly code
scdg_ann	Annotate SCDG with basic block and profile information
scdg_sti	Performs profitability analysis and SCDG transformations
scdg2c	Convert SCDG to C code

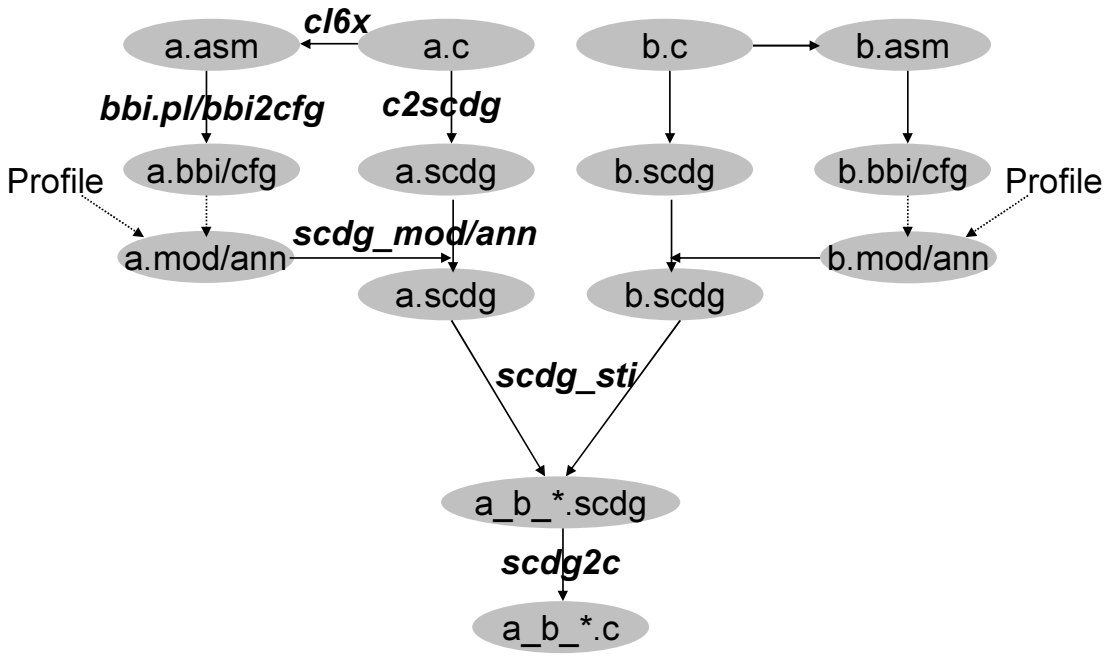


Figure 5.1: STI for ILP Tool Chain overview

also supports printing the complete alignment set with corresponding estimated profitability (EP) and code size increase, and generating a decision tree in the DOT language.

5.3 Operation of STI for ILP Tool Chain

5.3.1 SCDG Construction

The first step for integration is construction of SCDGs (Simplified Control Dependence Graphs) from the C source code. Instead of writing our own C language parser, we choose to use the SUIF2 (Stanford University Intermediate Format) compiler [1]. SUIF itself is an intermediate program representation defined by programming primitives and object hierarchy, which is equivalent to the program written in C or Fortran. The SUIF2 compiler provides the tools to convert C source code to SUIF (*c2suif*) integrated with EDG front-end, and SUIF to C (*suif2c*). Any source level transformations can be implemented by a SUIF 'pass', which transforms one SUIF representation to another. A SUIF pass can be written by programmers with the help of various utility functions provided with the SUIF compiler.

We implement our own SUIF pass to construct SCDGs, called *scdgpass*. In this pass, loop, conditional and call statements are identified and appropriate SCDG objects are created accordingly. At the end of the pass, SCDG objects are stored into a binary file, which is fed into the program which performs SCDG transformation later. Figure 5.2 shows an example of SCDG automatically generated from C source code, and converted into the DOT language by the *scdg2dot* program.

To support code generation at the back-end after SCDG transformation, the C code statements are always stored along with each SCDG object. The SCDG can be converted into C code via the SCDG-to-C converter (*c2scdg*), that emits the C code which corresponds to each SCDG object by in-order tree traversal of the SCDG. Since only structured code is processed by our front-end, the source code must be converted into a structured form if the original code contains a non-structured component such

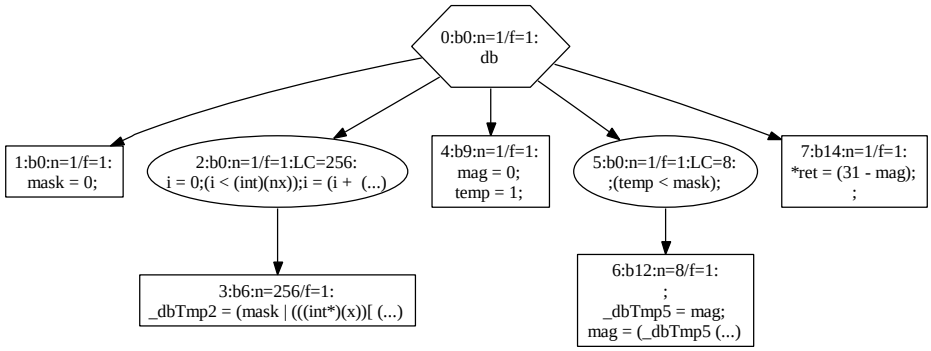


Figure 5.2: Example of automatically generated SCDG (DB)

as continue, break or switch-case statements.

5.3.2 Alignment Set Definition and SCDG Transformation

After construction of both SCDGs, alignment set definition begins with extracting code nodes from them. Definition of the alignment set (i.e. STI scenario space) is implemented by constructing a decision tree because an integration scenario is determined by the series of decisions which nodes to be integrated. Every child node in the decision tree corresponds to a possible distinct decision, which follows decisions taken to reach its parent node. Therefore, each node in the decision tree corresponds to a certain code alignment, which includes all combinations chosen to reach it from the top node. Figure 5.3 and 5.4 respectively show the example decision tree and the full list of alignments automatically generated for one example case (DB + IH), one of the procedure pairs we examine in Section 5.4.

At the initial stage of decision tree construction, every combination of code nodes creates a distinct decision tree branch because any decision is possible at the beginning. Once one combination is chosen, not all combinations are possible because only code combinations which do not violate the data and control dependencies are allowed. This is the concept of 'legal' alignments discussed in Section 4.2.3. Identifying legal alignments is implemented by associating creation of a decision tree node with hierarchical SCDG transformation.

Integration of two threads is achieved by merging two SCDGs into one. A certain

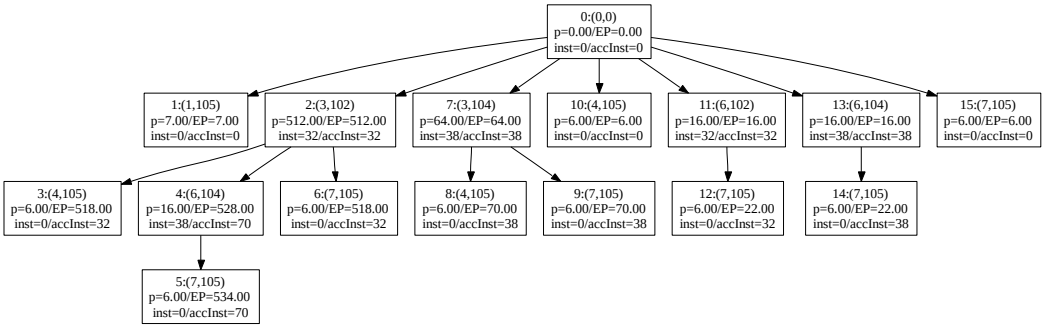


Figure 5.3: Example of automatically generated decision tree (DB + IH)

ID	EP	Inst	Alignment set
1	7.00	0	{ (a1 ,b105),}
2	512.00	32	{ (a3 ,b102),}
3	518.00	32	{ (a3 ,b102), (a4 ,b105),}
4	528.00	70	{ (a3 ,b102), (a6 ,b104),}
5	534.00	70	{ (a3 ,b102), (a6 ,b104), (a7 ,b105),}
6	518.00	32	{ (a3 ,b102), (a7 ,b105),}
7	64.00	38	{ (a3 ,b104),}
8	70.00	38	{ (a3 ,b104), (a4 ,b105),}
9	70.00	38	{ (a3 ,b104), (a7 ,b105),}
10	6.00	0	{ (a4 ,b105),}
11	16.00	32	{ (a6 ,b102),}
12	22.00	32	{ (a6 ,b102), (a7 ,b105),}
13	16.00	38	{ (a6 ,b104),}
14	22.00	38	{ (a6 ,b104), (a7 ,b105),}
15	6.00	0	{ (a7 ,b105),}

Figure 5.4: Example of automatically generated alignment set (DB + IH)

code transformation is applied depending on control dependence and estimated profitability of the target nodes (i.e. code combination). Once target nodes are chosen, transformations are applied hierarchically from the the top nodes to the target nodes. In every hierarchy level, the SCDG is divided into left, center and right subgraphs in terms of the location of target nodes. All possible code combinations from left and right subgraphs determine the set of the combinations, that can coexist with the previously selected code combination in a legal alignment.

Figure 5.5 shows an example of hierarchical SCDG transformation. The SCDG on the top is the initial SCDG constructed by merging nodes from both procedure A and B, which names start with 'a' and 'b' respectively. Let's assume that the code combination (a_4, b_3) (shaded nodes in the figure) is selected for integration. At the top level hierarchy under the procedure node, the SCDG is divided into left, center and right subgraphs. All combinations from left and right subgraphs can coexist with (a_4, b_3) because they do not violate the data dependencies of original code. For example, the alignment $\{(a_1, b_1), (a_4, b_3)\}$ is legal and so is $\{(a_4, b_3), (a_8, b_5)\}$. At the second level hierarchy under the node a_2, another right subgraph is created. The code combination from this subgraph can also coexist with (a_4, b_3) as the alignment $\{(a_4, b_3), (a_6, b_3)\}$ is legal. Since every subgraph is nothing but another SCDG, decision tree construction and SCDG transformation are repeated in a recursive manner until all possible alignments are found. This enables full enumeration of the alignment set. Figure 5.6 shows the algorithm for hierarchical SCDG transformation and alignment set definition. If the function *PerformSTI* is called once with the initially merged SCDG, it creates a complete decision tree by recursively calling itself with right subgraphs generated during hierachical SCDG transformation.

5.3.3 Full vs. Greedy Enumeration

Our implementation is based on full enumeration of alignments by evaluating every single possible combination at every decision tree node in a recursive manner. For each node construction, the profitability is estimated and the SCDG is transformed. Full enumeration builds the complete decision tree, where each node has the

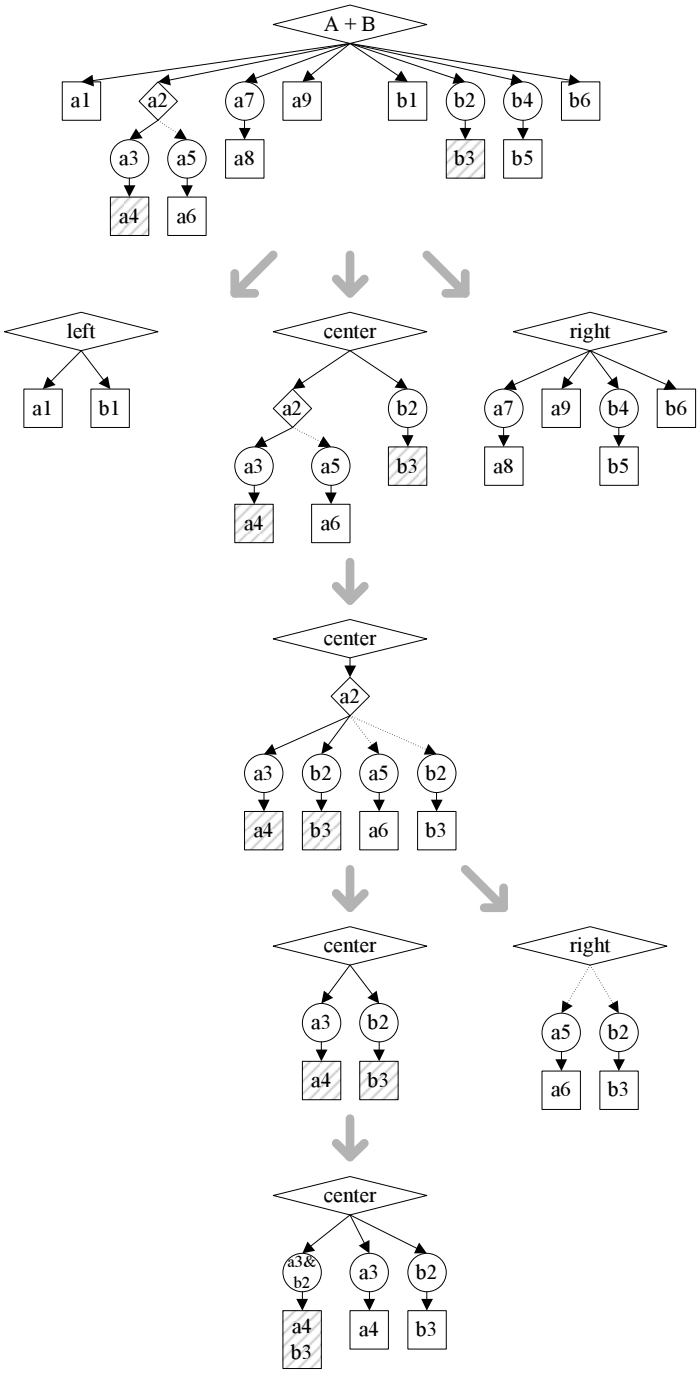


Figure 5.5: Example of hierarchical SCDG transformation

```
procedure PerformSTI(scdg, parentDT)
for all code combination (a_i, b_j) in scdg do
  nodeDT ← create a new decision tree node under parentDT;
  n_a ← top parent SCDG node of a_i in scdg;
  n_b ← top parent SCDG node of b_j in scdg;
  repeat
    scdgR ← Aggregate n_a and n_b then obtain the right subgraph from scdg;
    call PerformSTI(scdgR, nodeDT);
    if n_a is a predicate then
      duplicate n_b to every case of n_a;
      n_a ← next level SCDG node of a_i in scdg;
    else if n_b is a predicate then
      duplicate n_a to every case of n_b;
      n_b ← next level SCDG node of b_j in scdg;
    else if both n_a, n_b are loops then
      apply loop jamming and unrolling for n_a and n_b;
      n_a ← next level SCDG node of a_i in scdg;
      n_b ← next level SCDG node of b_j in scdg;
    end if
  until n_a = a_i and n_b = b_j
end for
```

Figure 5.6: Algorithm for hierarchical SCDG transformation and alignment set definition

corresponding EP and integrated SCDG. Though both pseudo-scheduling and SCDG transformation are fast processes, full enumeration of alignments can be expensive in terms of both time and memory space because the upper bound of the alignment set size increases exponentially as number of code nodes increases. Therefore, full alignment enumeration would not be effective in the cases where both threads include many code nodes.

An alternative to full enumeration is greedy enumeration. Instead of evaluating all possible combinations from each decision tree node, one combination which is estimated the most beneficial can be chosen. For the example decision tree in Figure 5.3, the second branch would be chosen at the first level because it has the largest profitability value (p=512). At the next level, the second branch (p=16) would be selected. Greedy enumeration hence evaluates Alignment 5 only, which happens to be the alignment with the maximum EP among all in this case. However, greedy enumeration does not guarantee reaching the alignment with maximum EP due to its nature.

Therefore, hybrid approaches are desirable in practice rather than pure greedy enumeration. Instead of selecting only one combination at each decision tree node, a fixed number of combinations with higher EPs or ones with EPs lager than a certain threshold can be evaluated. Another possible approach is applying different schemes according to code node types. Since looping code has more impact on the performance, we can enumerate full alignments from looping code combinations but apply greedy enumeration scheme for acyclic code combinations. These approaches would save time for searching the best estimated alignment despite possibility of reaching a sub-optimal point. Examination of these approaches is left as future work.

5.3.4 Code Size Increase Estimation

Since most STI transformations involve code duplication, code size increase is another important factor to be considered. It could affect the run-time on a real processor depending on instruction cache sizes and configurations, as observed in the experimental results on an Itanium processor presented in Chapter 2.

Code size increase estimation is tightly coupled with SCDG transformation because which code nodes are duplicated is determined at this stage. For code size increase estimation, we use instruction count information of the basic block annotated into each code node. Every time a code node is duplicated, we accumulate estimated instruction count increase. By doing so, every alignment has a corresponding estimated instruction count increase as well as its EP. The 'Inst' column in Figure 5.4 shows the estimated instruction count increase of each alignment for the example case. For the TI C6x architecture, an instruction count metric can be converted into a byte metric by multiplying 4 because every instruction is 32-bit wide.

As static profitability estimation is more optimistic than the real scheduling, this code size increase estimation predicts a smaller than actual code size increase. While estimation only considers the code nodes duplicated in terms of SCDG representation, there are more instructions generated due to added conditional statements and side effect code. This impact is especially large in case of software pipelined (SWP) loops because the compiler often generates a non-SWP prolog loop for covering the case where the loop count is less than the minimum loop count required by an SWP loop.

In order to take this kind of overhead into account, the upper bound for code size increase is determined by multiplying the estimated code size increase by a fixed overhead factor. The overhead factor is determined empirically; We use 2 or 4 in our experiments.

Though we use only performance measures to select the best integrated procedure in this work, code size estimation can be considered for a better decision. The simplest example is filtering out the integrated threads, where estimated code size exceeds the instruction cache size. This is critical if the architecture features an instruction cache. Advanced techniques for balancing the performance and code size such as [118] can be also applied when choosing the best integrated procedure.

Table 5.2: Summary of experimental cases

Procedure pairs	Proc.A R_A	Proc.B R_B	Alignments L_{AB}	Best speedup	Code size increase
(a) DB + IH	5	3	15	6.57%	38.9%
(b) XF + BR	8	3	39	7.61%	44.7%
(c) BCE + BSK	6	7	48	9.50%	34.1%
(d) AC + CLTP	13	17	5697(21*)	12.46%	54.8%
(e) XF + IH	8	3	45	9.91%	22.0%
(f) XF + DB	8	5	192	9.45%	46.2%

*: Loop only

5.4 Experiments and Results

For the experiments, we use the same target architecture (TI TMS320C64x), compiler and evaluation methods as the experiments in Chapter 4. The same 8 procedures extracted from TI DSP/Image Libraries and MiBench are used for integration, but the C code for integrated procedures is automatically generated by the tool chain, unlike the experiments in the previous chapter. In addition to 4 pairs of procedures, 2 more cross combinations are added to examine capability of the tool. We call the experimental cases (a) through (f) for consistency with Chapter 4.

Table 5.2 summarizes number of code nodes, alignments for procedure pairs, the speedup of the best integrated procedure and its code size increase. Compared with the experiments in the previous chapter, the alignments set sizes become significantly larger because alignments are constructed from both cyclic and acyclic code combinations by the help of automatic tool chain. Case (d) is evaluated only for the alignments constructed from cyclic code node combinations because there are too many alignments. Figure 5.7, 5.8, 5.9, 5.10, 5.11, and 5.12 show the estimated and actual speedup (graph (a)), the estimated and actual code size increase (graph (b)) for 6 pairs of procedures respectively. Alignments (i.e. integrated procedures) are ordered by their initial estimated speedups from the left to the right.

The estimated speedups are computed based on their initial EP values by converting a cycle metric into a speedup metric. Actual speedups are measured by executing integrated procedures on the TI C6x simulator. The points emphasized with red

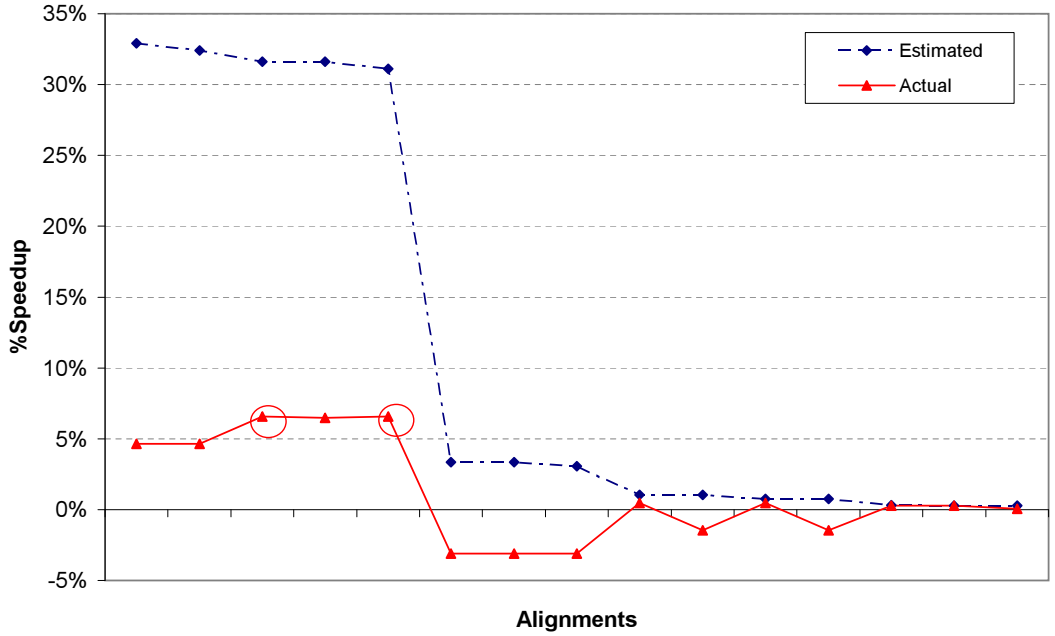
circles indicate the procedures showing the best speedups among all integrated procedures. In code size increase graphs, two estimated values are plotted based on two overhead factors, 2 and 4. Actual code size increases are computed by measuring the code sizes of original and integrated procedures.

From speedup graphs, we are interested in two facts: (1) how accurate the initial estimation is and (2) how fast the actual best alignment can be reached by the iterative compilation algorithm. For the first, we compare the overall trend of estimated speedups with actual speedups. For the second, we need to determine how many compilation cycles are required by examining the details of alignments.

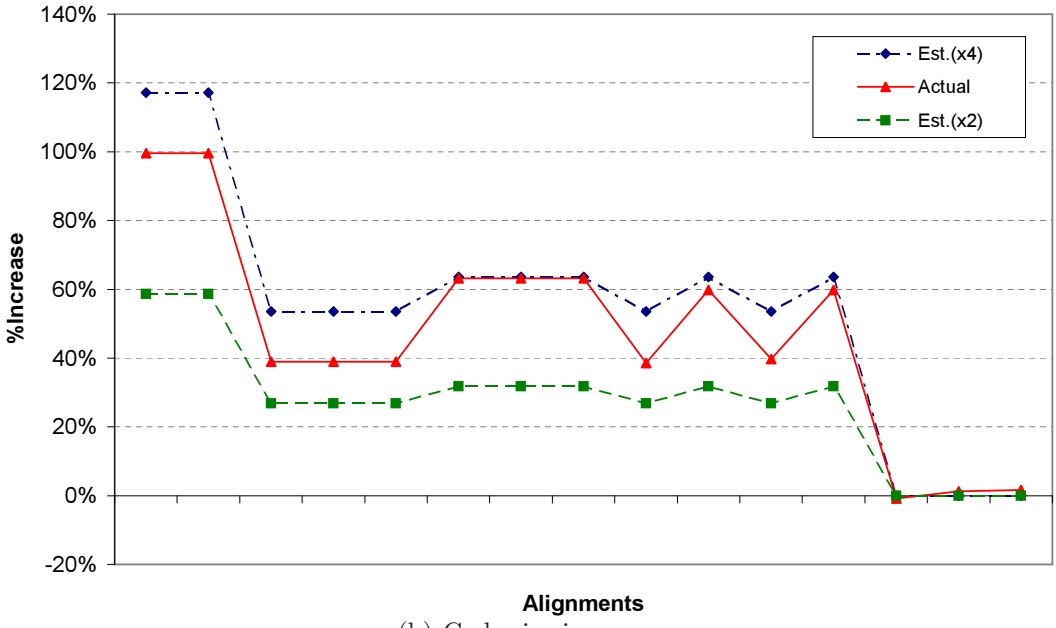
In case (a), estimated speedups show a decreasing trend similar to the actual speedups. Estimation errors are mainly from one loop combination, which turns out to be non-beneficial after compilation. Since first two alignments include this loop combination and the other (beneficial) loop combination, the EPs of these two alignments would become lower after compilation of the first alignment. Therefore, the iterative compilation method reaches the third alignment (i.e. the best integrated procedure) after one compilation cycle. The following two alignments (fourth and fifth) contain the same loop combination but different acyclic code combinations. Since acyclic code combinations usually have little performance impact, the acyclic code alignment can be chosen based on the initial EP value without iterative compilations.

In case (b), estimated speedups show a similar trend to the actual speedup except for the first six points. These six alignments include the same loop combination where the actual SWP schedule is much worse than what is estimated. After compilation of the first alignment, the iterative compilation method simultaneously updates the EPs of these six alignments. Then the seventh alignment is selected as it has the maximum EP. The best alignment (eighth) is reached after another compilation cycle but compilation continues until the rest of loop alignments are examined because their EPs are equal, as observed in Section 4.4.

In case (c), the best integrated procedure is found at the first compilation cycle because the best alignment is the one with the largest estimated speedup. Nevertheless, the estimated speedups do not show consistency with actual speedups because pseudo-scheduling errors from code nodes with subroutine calls are large, as discussed

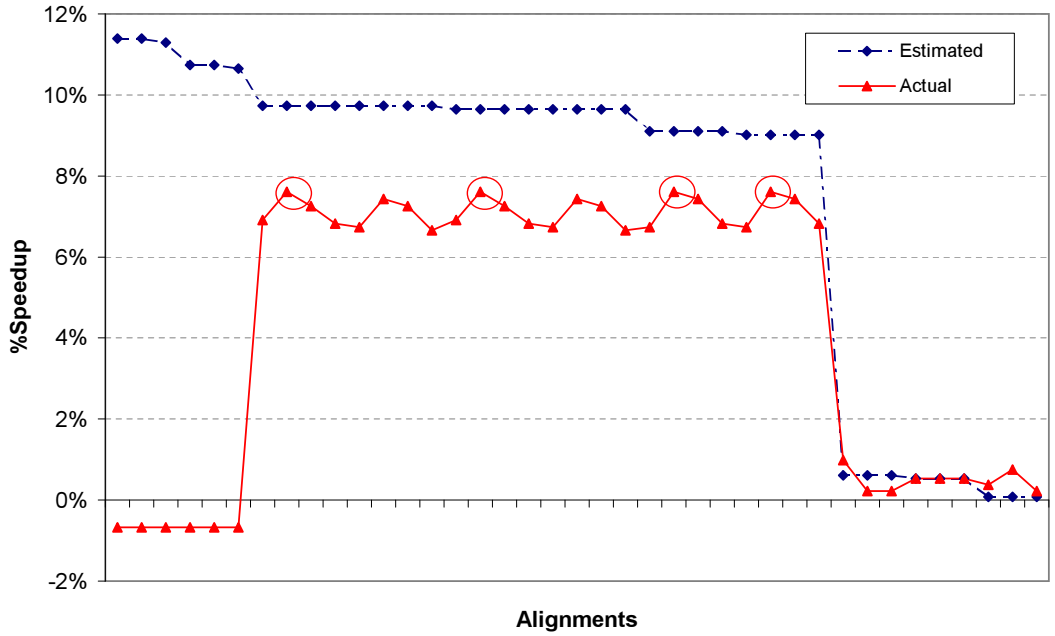


(a) Speedup

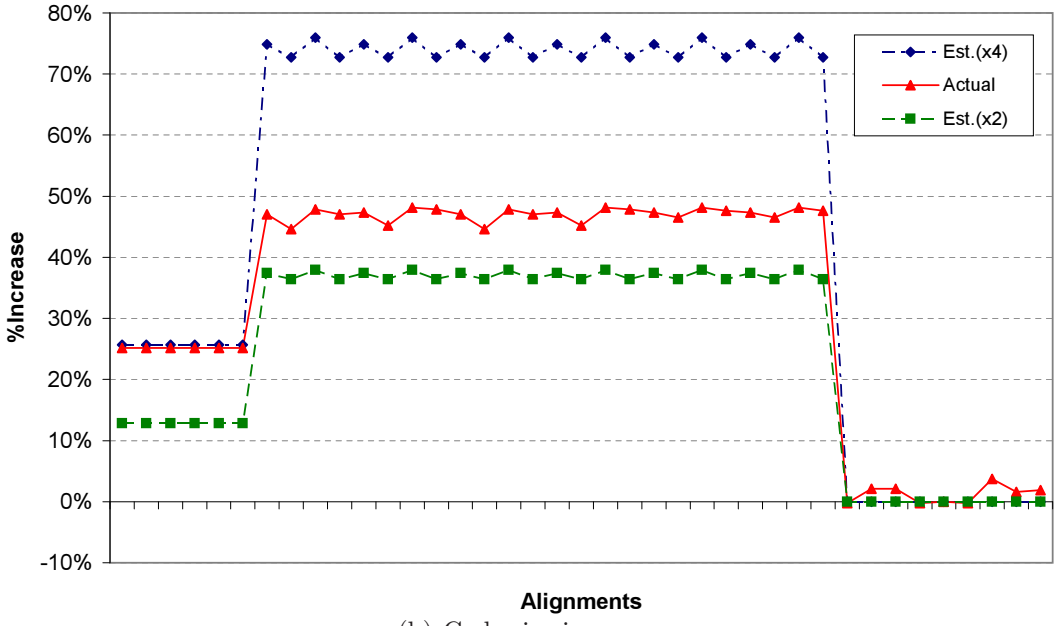


(b) Code size increase

Figure 5.7: Case (a) DB + IH: Estimated/actual speedup and code size increase

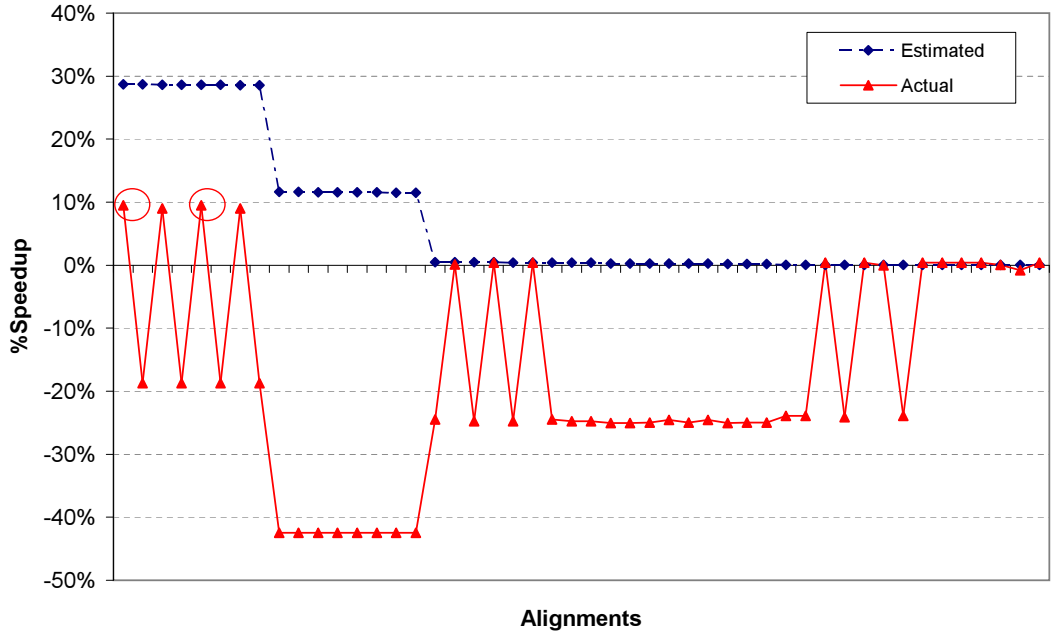


(a) Speedup

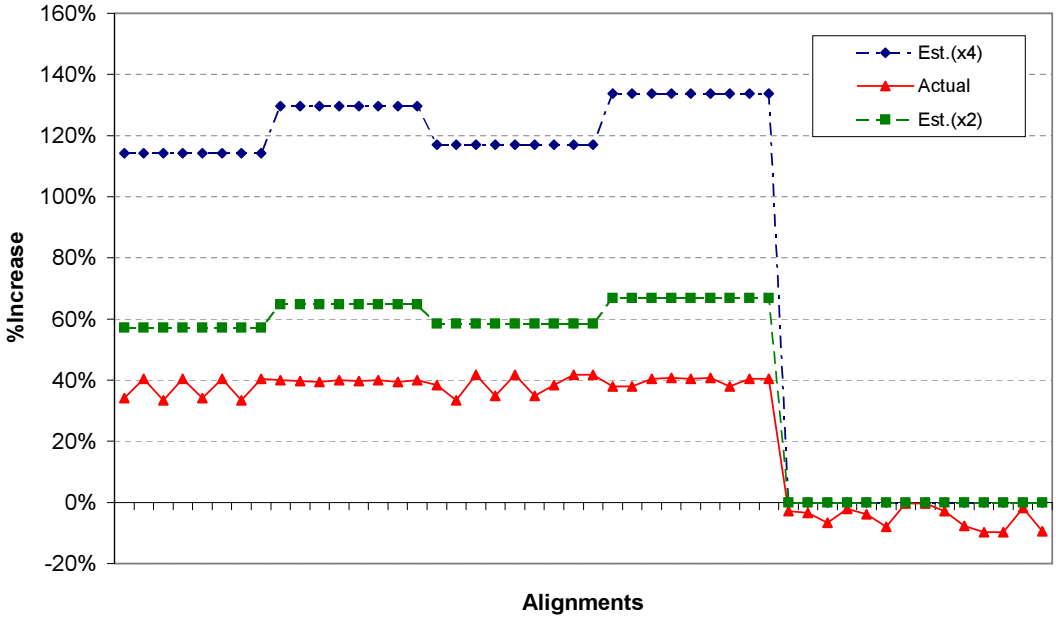


(b) Code size increase

Figure 5.8: Case (b) XF + BR: Estimated/actual speedup and code size increase

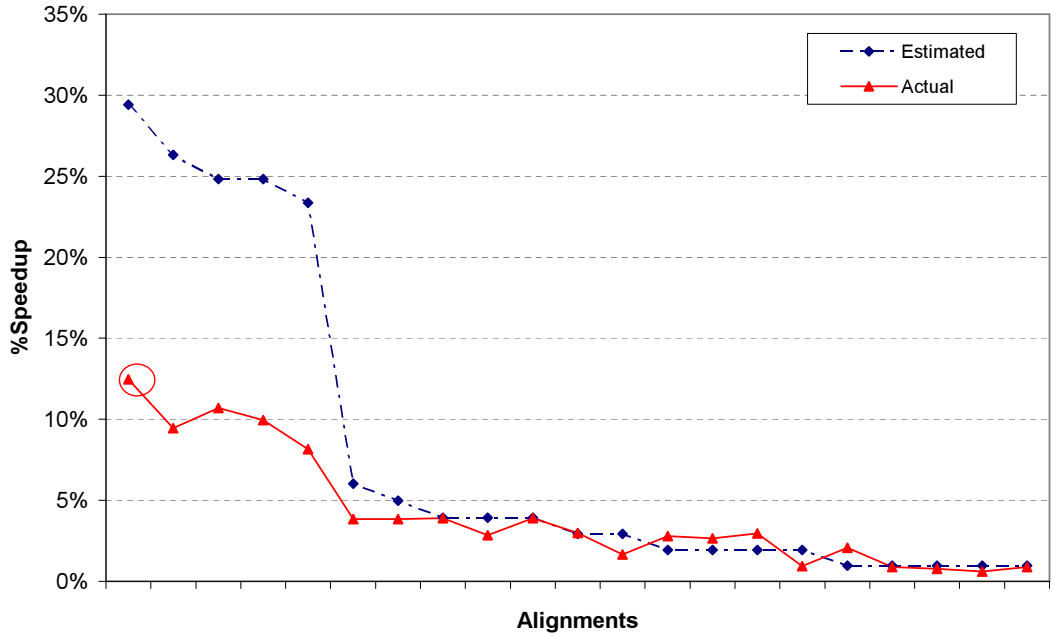


(a) Speedup

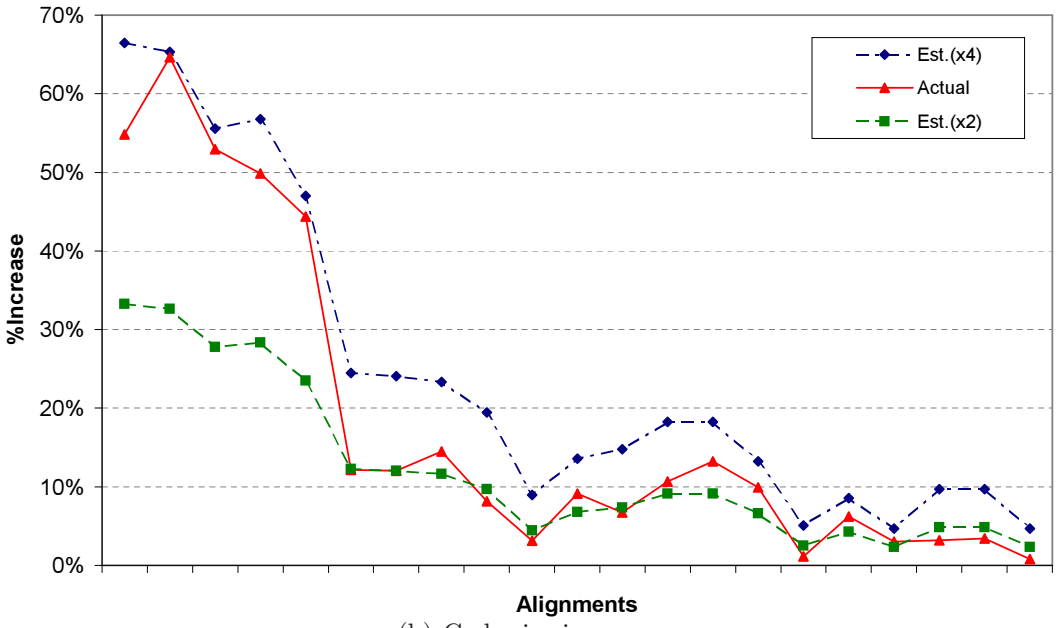


(b) Code size increase

Figure 5.9: Case (c) BCE + BSK: Estimated/actual speedup and code size increase

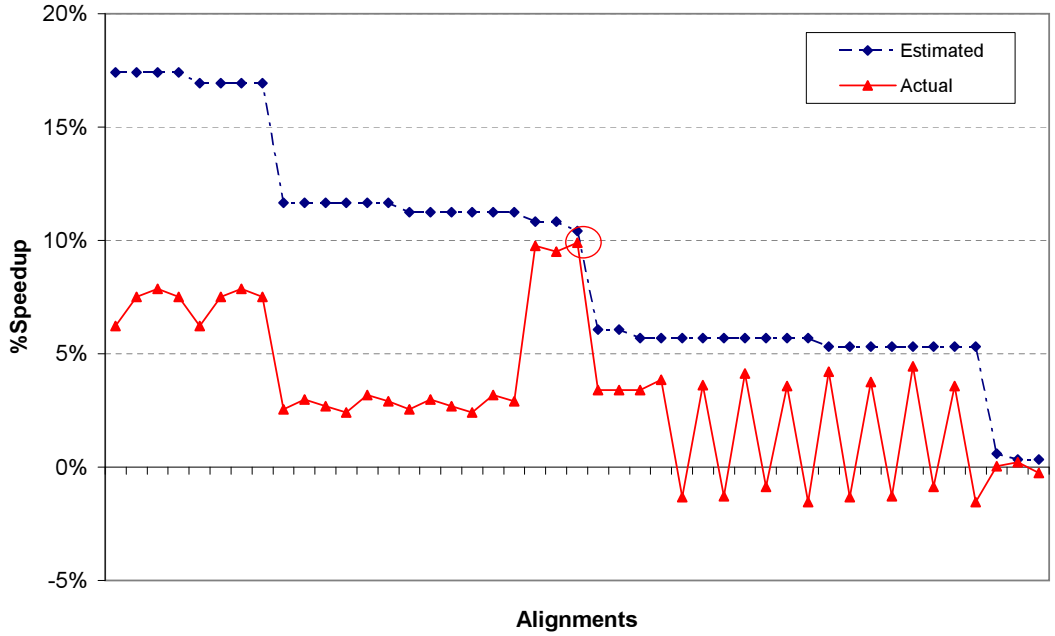


(a) Speedup

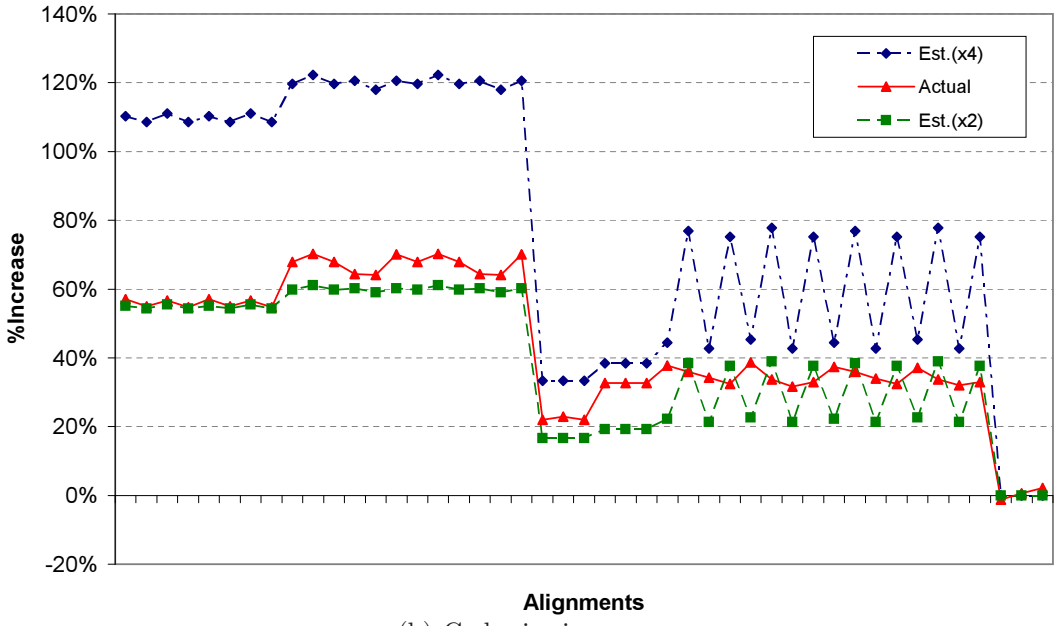


(b) Code size increase

Figure 5.10: Case (d) AC + CLTP: Estimated/actual speedup and code size increase

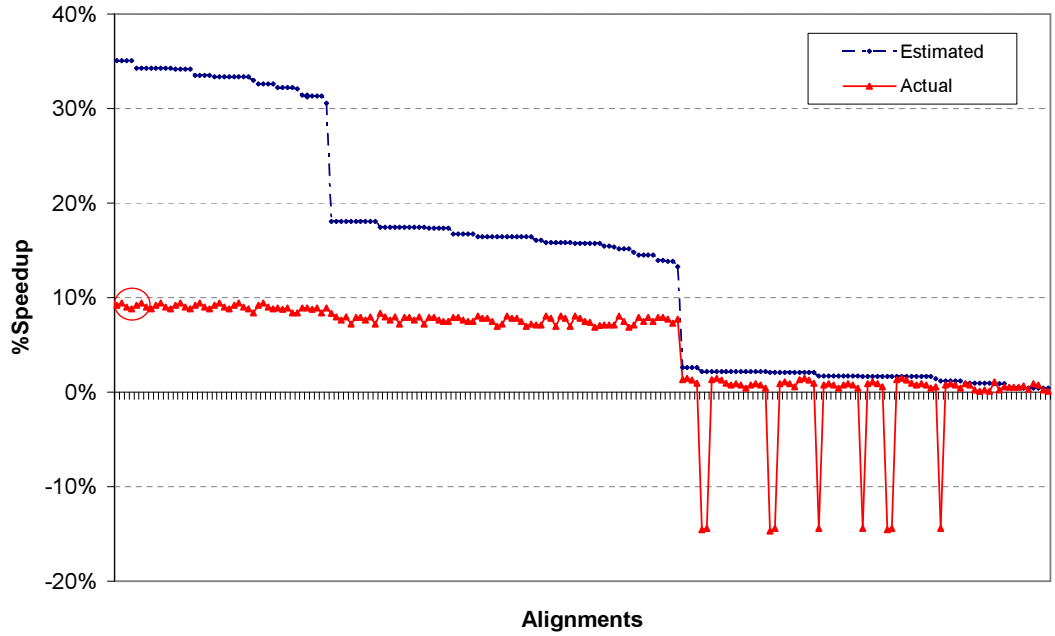


(a) Speedup

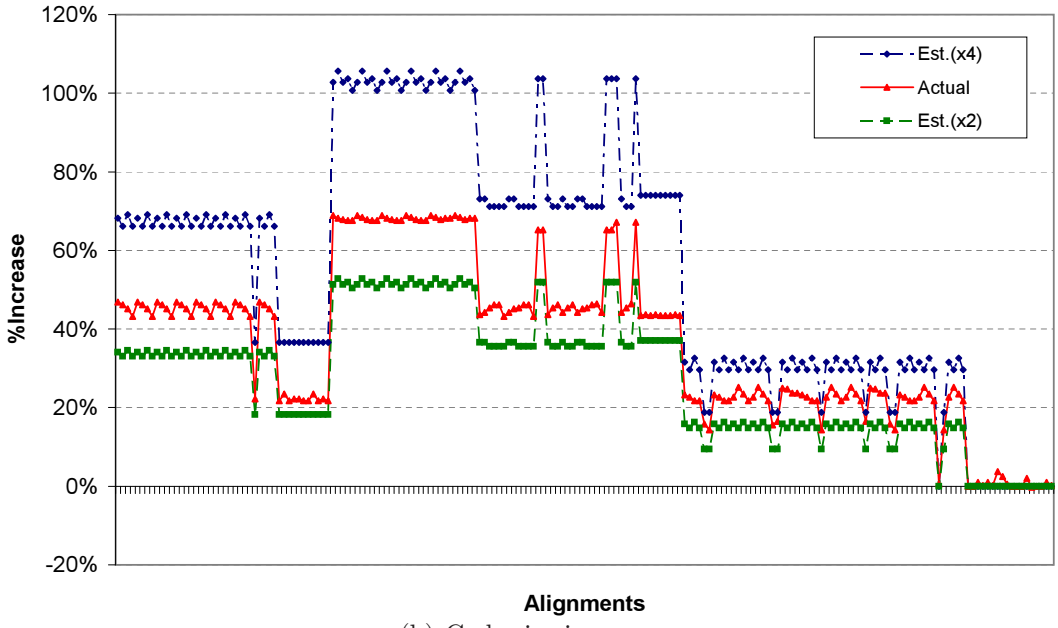


(b) Code size increase

Figure 5.11: Case (e) XF + IH: Estimated/actual speedup and code size increase



(a) Speedup



(b) Code size increase

Figure 5.12: Case (f) XF + DB: Estimated/actual speedup and code size increase

in Section 4.4. In addition, there is another type of error in this case. The first 8 points in Figure 5.9(a) correspond to the alignments which include the best cyclic code combination but different acyclic code combinations. Though a small speedup variation is expected, it shows a huge actual speedup difference. It turns out that some acyclic code affects the schedule of other code depending on where it is located. One explanation is that code interleaving during STI can affect register allocation. If we insert a code node between two consecutive code nodes, the registers live between two consecutive code nodes are not available for the middle node. The instruction schedule of the middle node can become worse after integration if its original register pressure is heavy. If the schedule of the non-integrated code becomes significantly worse after STI, different acyclic code alignments must be used to avoid such side effects, though it does not happen frequently.

In case (d), we construct alignments only from cyclic code nodes. The overall trend of actual speedups is consistent with the estimated trend, indicating estimation works well. The best alignment is found after the first compilation cycle. Some alignments have higher actual speedups than estimated speedups due to the impact of induction variable elimination.

In case (e), the estimated speedups show some difference with the actual speedups. Four different loop combinations included in the alignments located on the first and second estimated speedup plateaus (first twenty alignments) turn out to be non-beneficial after compilation. Since the first four alignments include these loop combinations, the EPs are updated quickly after four compilation cycles. Then, the iterative compilation method finds the alignments with the best loop combinations (three alignments in the actual speedup peak).

Case (f) shows a good relationship between estimated and actual speedups as they show the same decreasing trends. The overall actual speedup order is consistent with the estimated order despite some estimation errors for the first alignments. The best integrated procedure is also found with the very first few alignments.

Code size increase estimation works quite well for most cases as the estimated and the actual code size increases show similar trends. The actual code size increases fall between overhead factors of 2 and 4 except case (c) where these is no software

pipelined loop. Though code size increases and speedups do not show direct correlations, the best integrated procedures involve at least 20% code size increases.

5.5 Future Extensions

The following extensions can be added to the current STI for ILP Tool Chain for more applicability:

- The step for SCDG modification and annotation can be automated. This is the only manual step in the current tool chain, where users write modification and annotation directive files (.mod/ann) by comparing the CFG and the SCDG. Automation of this step requires extension of the assembly processing tools so that they match basic blocks with the corresponding C source code parts based on target compiler's code generation rules and the information embedded in the assembly code. Since the same technique can be used to match the integrated SCDG with the assembly code generated after each compilation cycle, it would fully automate the feedback path of the iterative compilation method, enabling complete automation of our proposed methods.

- Additional optimizations for integrated code can be built into the tool chain, such as induction variable elimination and loop invariant code motion. As discussed in Section 4.2.5, the impact of these optimizations can be significant when the integrated loop is tightly scheduled.

- Subroutine calls can be included in analysis and transformation as discussed in Section 4.2.7. This will extend the applicability of STI for ILP to threads with arbitrary sizes, which may contain nested subroutine calls.

- The tool chain can be extended to deal with (partially) dependent thread as discussed in Section 4.2.7. Data flow analysis must be build into the tool for data dependent threads. This enables STI for ILP applicable to threads with data dependencies or timing constraints.

5.6 Conclusions

We have created the tool chain, *STI for ILP Tool Chain* to automate most of the work in STI for ILP, making this technique more convenient and useful. Our tool chain converts the C source code of original threads into hierarchical graph representations. After the instruction schedule information from the assembly code compiled for a target processor (TI C6x in this case) is annotated into them, our tool automatically creates the set of possible integration scenarios along with the hierarchical graph transformation required. Then, integrated procedures are automatically generated from the transformed graph representations along with their estimated speedup and code size increase.

By automatically integrating procedures from the TI DSP/Image Libraries and MiBench by the tool chain, we have observed that our estimation schemes for speedup and code size increase work well.

Chapter 6

STI for Streaming Applications

6.1 Introduction

Being coarse-grain (procedure-level) dataflow representations, stream programming languages explicitly provide the data dependence information which constrains the selection of which code to integrate. In this chapter we investigate the integration of multiple procedures (or threads) in streaming applications on VLIW processors to improve performance through better processor utilization.

Among many languages implementing dataflow paradigms, we use StreamIt [106], a C-like stream programming language developed at MIT. The StreamIt program representation provides a good platform for applying STI for three reasons. First, it fully exposes parallelism between filters via its dataflow representation. Second, the StreamIt compiler supports a uniprocessor back-end by generating C code from StreamIt programs. Lastly, a single filter in a StreamIt program is converted into a single procedure in the C program, so the granularity of parallelism expressed in a StreamIt program exactly matches what is desired for STI.

In a StreamIt program, the filters are described with C-like language with *push*, *pop* and *peek*, by which they communicate with adjacent filters. The dataflow between

filters is expressed via constructs – *Pipeline*, *SplitJoin* and *FeedbackLoop*. A StreamIt program can be represented as a stream graph, where arcs defines use-def chains of data. Figure 6.1 and 6.2 show an example StreamIt application *BPFProgram* and the corresponding stream graph, which is automatically generated by the StreamIt compiler. (See Section 1.5.4 for an overview and [95] for more detailed features of the StreamIt language and compiler.)

We have developed methods called *StreamIt filter integration* to apply STI at the StreamIt level. The methods include identifying, selecting and integrating filters (i.e. procedures) from StreamIt applications. We also propose the techniques to transform programs and optimizations for targeting a VLIW DSP. An example StreamIt application is integrated via filter integration and the C code for the integrated program is generated by the StreamIt compiler uniprocessor back-end. We then compile the C code for the Texas Instruments C67x VLIW DSP architecture and evaluate its performance using the TI C6x simulator.

This chapter is organized as follows. Section 6.2 describes the techniques used to determine which filters to integrate, how to modify the stream graph, and optimizations for uniprocessors. Section 6.3 presents the experimental method and results. Section 6.4 summarizes the findings.

6.2 StreamIt Filter Integration

We use the dataflow information expressed by the StreamIt language to identify parallel filters. The candidate filters are chosen based on profiling or run-time estimation. Integrated filters are generated by the STI code transformations described in previous chapters. Finally, graph transformation is followed to transform the application to use integrated filters.

```
float->float pipeline BandPassFilter (float rate, float low, float high, int taps) {
  add BPFCore(rate, low, high, taps);
  add Subtracter();
}
float->float splitjoin BPFCore (float rate, float low, float high, int taps) {
  split duplicate;
  add LowPassFilter(rate, low, taps, 0);
  add LowPassFilter(rate, high, taps, 0);
  join roundrobin;
}
float->float filter Subtracter {
  work pop 2 push 1 {
    push(peek(1) - peek(0));
    pop(); pop();
  }
}

float->float filter LowPassFilter(float rate, float cutoff,
                                  int taps, int decimation) {
  float[taps] coeff;
  init {
    int i;
    float m = taps - 1;
    float w = 2 * pi * cutoff / rate;
    for (i = 0; i < taps; i++) {
      if (i - m/2 == 0) coeff[i] = w/pi;
      else coeff[i] = sin(w*(i-m/2)) / pi / (i-m/2) * (0.54 - 0.46 * cos(2*pi*i/m));
    }
  }
  work pop 1+decimation push 1 peek taps {
    float sum = 0;
    for (int i = 0; i < taps; i++)
      sum += peek(i) * coeff[i];
    push(sum);
    for (int i=0; i<decimation; i++)
      pop();
    pop();
  }
}

void->void pipeline BPFProgram {
  add FloatSource();
  add BandPassFilter(250000000, 1000, 1200, 64);
  add FloatPrinter();
}

void->float filter FloatSource {
  float x;
  init { x = 0; }
  work push 1 {
    push(x);
    x = x + 1.0;
  }
}

float->void filter FloatPrinter {
  work pop 1 { print(pop()); }
}
```

Figure 6.1: Example StreamIt application, *BPFProgram* [94]

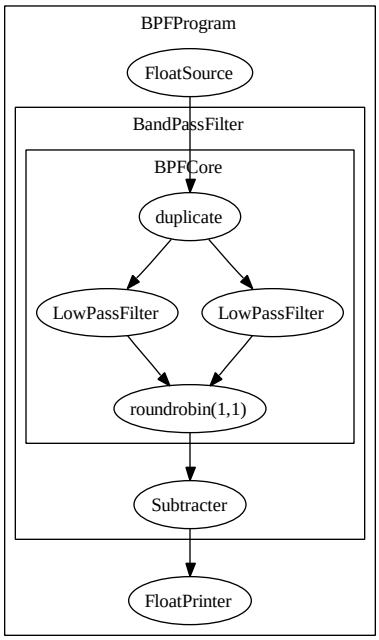


Figure 6.2: Stream graph of *BPFProgram*

6.2.1 Identifying the Candidate Filters

In StreamIt programs, data-independent filters exist in a *SplitJoin* region, which is defined by a *splitter*, a *joiner* and filters between them. Any filters not connected with each other in a *SplitJoin* region can run in parallel therefore the candidates for integration. By profiling the application or estimating run-time of each filter, the filters dominating the performance of the application are identified.

6.2.2 Filter Code Transformation

The code inside filters in StreamIt programs is converted into actual C procedures. This code is almost same as the normal C code except that it takes input and output data by *peek*, *pop* and *push* instead of procedure arguments. Since the StreamIt express control flow in the same way as C, the code transformation techniques proposed in previous chapters can be applied accordingly. *Peek*, *pop* and *push* are converted into memory accesses and a pointer increment (or decrement) – *peek*, *pop* perform

memory reads while *push* performs a memory write – in actual procedures. Hence, they are treated as memory accesses during code transformation. Though the real references are not shown in the language, they can be identified with analysis.

By performing dataflow analysis on top of it, redundant code can be eliminated during integration. For example, *pops* following a *duplicate splitter* retrieve the same data element hence only one *pop* is needed in the integrated filter. On the contrary, *pops* following a *roundrobin splitter* retrieve the different data elements hence two distinct *pops* are needed in the integrated filter. Since the parameters of filters are determined statically by other parts of StreamIt code, they can be used to optimize transformations. For example, loop trip counts determined statically, can remove the need for clean-up loops in some cases.

6.2.3 Graph Transformation

After writing integrated filters, the stream graph of the application is modified so that it can run integrated filters. The graph transformation is achieved by modifying the part of the StreamIt code other than filters. As any arbitrary combinations of data-independent filters in a *SplitJoin* region are possible for integration, any corresponding graph transformations are feasible as well. The graph transformations not only merge filters in a *SplitJoin* region into one filter but also divide one *SplitJoin* region into multiple ones by adding *splitters*, *joiners* and *identity filters* as needed, enabling a variety of transformations.

The filters in a stream graph are scheduled with a certain algorithm such as single appearance scheduling [39] and phased scheduling [60]. Though the scheduling algorithm minimizes the graph with necessary *splitters* and *joiners*, the transformed stream graph after use of integrated filters can be less efficient because of the overhead from additional *splitters* and *joiners*. Therefore, the balance in performance between integrated filters and the overall application is pursued. Existing graph transformations such as *fusion* and *fission* also are incorporated for efficiency.

6.2.4 Optimizations for Uniprocessors

Though the StreamIt compiler supports both the Raw processor and uniprocessor back-ends, its functionalities are more focused on parallelizing and load-balancing the application for communication exposed architectures [39]. Therefore, the code generated for uniprocessors does not show promising performance as compared with optimized C code. For instance, the C code generated by the StreamIt compiler is slower than the equivalent hand optimized C code in factor of 10 in case of GSM Decode [107]. This is because C code conversion process in the current StreamIt compiler implies huge inefficiency for uniprocessor platforms.

The first source of inefficiency is the unnecessary buffer allocation and memory copies caused by *splitters* and *joiners*. A *splitter* allocates multiple buffers for respective child nodes and copies input data items to the output buffers. A *joiner* also copies input data items of its parents to the output buffer. Buffer spaces containing the copied data items can be optimized into one buffer space. Memory copy operations can be eliminated by passing pointers.

The second is redundant code generation. The StreamIt compiler generates separate procedures for all instances of filters in the stream graph even though there are identical filters. This can be optimized so that the compiler generates separate procedures only for different filters but generates the same procedures for identical filters which can be shared by many instances.

The third is the granularity of filters. If a StreamIt program is composed of many filters with small amount of code, the compiler generates numerous procedures with a small number of instructions. This causes inefficiency because of procedure call overhead and buffers between filters. Forming appropriate granularity of filters by fusion transformation is desired to balance the code size and execution efficiency.

6.3 Experiments and Results

Filter integration is applied to one StreamIt application, *FMRadio* by applying manual code transformation to StreamIt source code. As shown in Figure 6.3, it is composed of series of hierarchical filters. FloatSource is a data source while FloatPrinter is a data sink. The main work of the program is the Equalizer, where input data are processed by 4 parallel BandPassFilters. BandPassFilter also include 2 parallel LowPassFilters connected with splitters and joiners.

For the first filter integration, we integrate two LowPassFilters inside the BPFCore creating a new integrated filter, BPFCore_Int2. During this integration, the loops in the two LowPassFilters are jammed because two loops have the same iteration counts. The splitter and joiner in the BPFCore are also removed after integration. Figure 6.4 shows the Stream graph after the first filter integration.

Since there are still parallel filters after the first integration, we apply another filter integration on them. There are four pipelines of filters with BPFCore_Int2, Subtractor, and AnonFilter_a18. Four identical filters in these pipelines are integrated together creating integrated filters, BFPCore_Int2_Int4, Subtacter_4, and AnonFilter_a47. During integration, loop jamming is applied to four loops in BFPCore_Int2 and some redundant operations are also removed. After the second filter integration, the stream graph is changed as shown in Figure 6.5. After this integration, there is no further filter integration is possible because there is no parallel filters.

For performance evaluation, we use Texas Instruments TMSC320C67x platforms supporting float data type operations because the application uses floating data types. The C source code is generated from these 3 different versions of FMRadio StreamIt code by the StreamIt compiler. The C code is compiled by the TI C6x compiler and the program is executed on the TI standalone C6x simulator. The execution cycles of filters of our interest are obtained by inserting cycle count APIs supported by the simulator.

Table 6.1 summarizes integration methods and speedups by filter integration. As shown in the execution profile of original *FMRadio* ('%Exec.' column), about 80% of the time is spent on running the EqSplit and more than a half is spent on run-

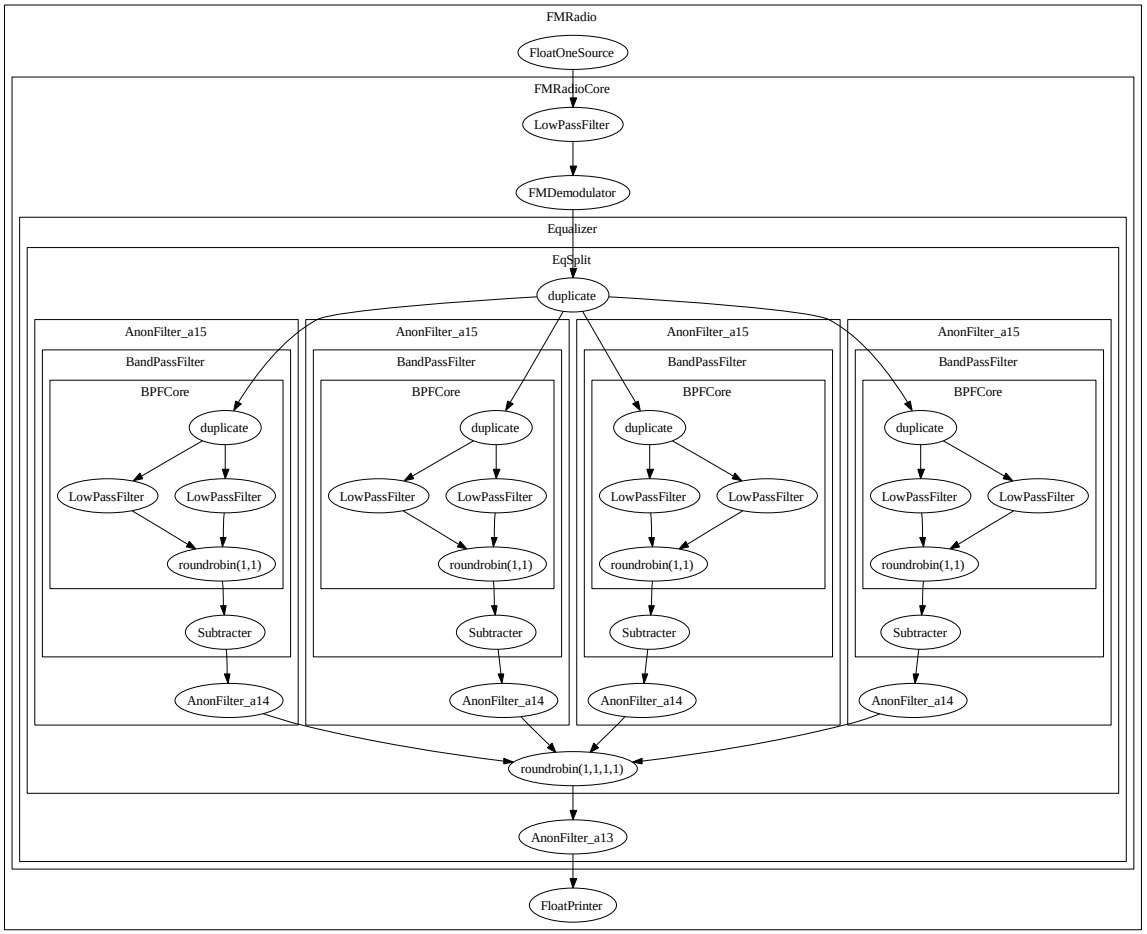


Figure 6.3: Stream graph of *FMRadio*

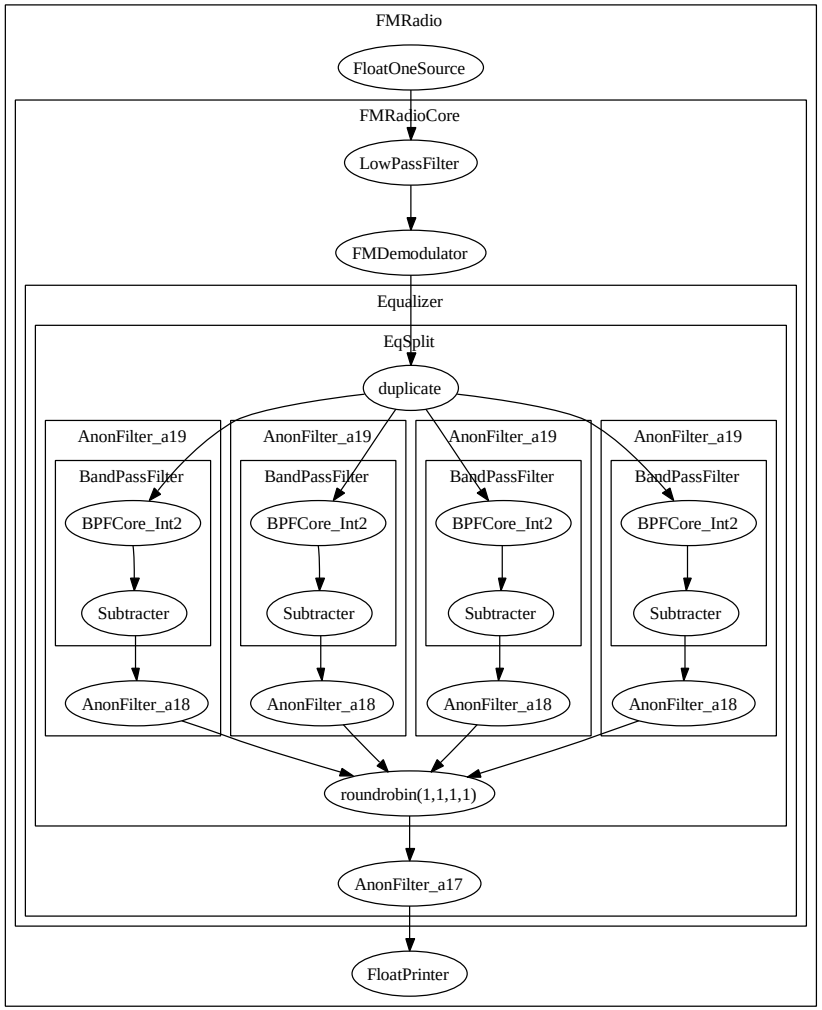


Figure 6.4: Stream graph of *FMRadio* after the first filter integration

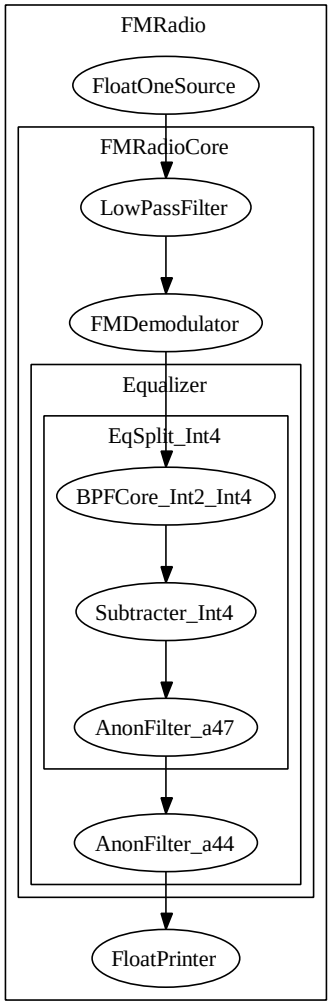


Figure 6.5: Stream graph of *FMRadio* after the second filter integration

Table 6.1: Speedup by filter integration of *FMRadio* on TI C67x DSP

Original filters	%Exec.	Integrated filters	Speedup	Integrated filters	Speedup
FMRadio	100%	FMRadio	1.80	FMRadio	3.40
FloatOneSource		FloatOneSource		FloatOneSource	
LowPassFilter		LowPassFilter		LowPassFilter	
FmDeModulator		FmDeModulator		FmDeModulator	
EqSplit	79.18%	EqSplit		EqSplit_Int4	9.21
splitter		splitter			
BPFCore×4	57.92%	BPFCore_Int2×4	4.23	BPFCore_Int2_Int4	10.52
splitter					
LowPassFilter×2	24.85%				
joiner					
Subtracter×4		Subtracter×4		Subtracter_Int4	
Filter×4		Filter×4		Filter_Int4	
joiner		joiner			
Filter		Filter		Filter	
FloatPrinter		FloatPrinter		FloatPrinter	

ning 4 sets of BPFCore. Integrating 4 sets of BPFCore (splitter + LowPassFilters×2 + joiner) gives 4.23 speedup in filter level and 1.80 in whole application level. By the second filter integration, BPFCore is sped up by 10.52 and EqSplit (splitter + BPFCore×4 + Subtracter×4 + Filter×4 + joiner) is sped by 9.21. Overall application-level speedup of 3.40 is achieved by this integration.

Huge speedups come from two sources. The first is the better compiler schedule and redundant code removal from the integrated filters by STI code transformations. The second is elimination of splitters and joiners by filter integration. Splitters and joiners in BPFCore are responsible for 33.08% of the execution time while 24.85% is spent on LowPassFilters as shown in Table 6.1. Since the time for running splitters and joiners is completely removed after integration, it has a huge impact on the overall performance. This is mainly due to inefficient uniprocessor code generation by the StreamIt compiler as discussed in Section 6.2.4. If the code is optimized for uniprocessors, the impact from splitters and joiners would be reduced.

6.4 Conclusions

In this chapter, we present the method called filter integration. It improves the performance of StreamIt applications by integrating parallel filters via STI. Parallel filters are identified by analyzing the dataflow information expressed by the StreamIt language. Integrated filters are generated by STI code transformation techniques. Accordingly, the program is transformed via graph transformation to take advantage of integrated filters.

We have applied filter integration to an example application, *FMRadio* and executed in on the TI C67x platform. Experimental results show that filter integration significantly helps StreamIt applications with targeting a VLIW uniprocessor DSP.

Chapter 7

Summary and Future Work

7.1 Summary

This dissertation proposes Software Thread Integration (STI) for Instruction Level Parallelism (ILP), a new way of converting thread-level parallelism into ILP on VLIW processors. STI improves the performance by merging parallel procedures into one, effectively increasing the compiler's scope and hence allowing it to create a more efficient code schedule. STI is essentially procedure jamming with intraprocedural code motion transformations which allow arbitrary alignment of instructions or code regions. This alignment allows code to be moved to use available execution resources better and improve the execution schedule. Parallel procedures are identified by the programmer with either annotations in procedural programming languages or graph analysis for stream programming languages, which represent programs as coarse-grain dataflow.

We introduce a method called procedure cloning and integration for improving program run-time performance by integrating parallel procedures via STI. The method creates specialized versions of procedures with better execution efficiency, providing clone procedures that do the work of multiple procedures concurrently through STI.

By a method called filter integration, we use STI for streaming applications, exploiting explicit dataflow information expressed by stream programming languages.

We define various STI code transformations to maximize the ILP and reconcile control flow differences between two threads. Different transformations are selectively applied according to the control structure and the ILP characteristics of the code, driven by interactions with software pipelining. This approach effectively combines ILP code transformations with instruction scheduling techniques so that they complement each other. Code transformations involve code motion as well as loop transformations such as jamming, unrolling, splitting, and peeling.

We propose a methodology for finding the best integration scenario efficiently among all possibilities. We introduce methods to quantitatively estimate the performance benefit from the integration of multiple software threads. We use resource modeling, consider register pressure and compensate for compiler optimizations. This enables different scenarios to be compared and ranked. We then use these estimates to guide integration by concentrating on the most beneficial scenario. Information from each iteration of compilation is used to update the rankings of scenarios. Our modeling methods combined with limited compilation quickly find the best integration scenario without requiring exhaustive integration. The proposed methods for STI for ILP are automated by the STI for ILP Tool Chain targeting Texas Instrument C6x VLIW DSPs.

7.2 Future Work

The concept of STI for ILP presented in this work encourages further investigations into STI and optimizations for digital signal processors. Many interesting research questions arise, such as:

- Is a fast estimation method with better accuracy feasible? Can the schedule of integrated code be estimated more accurately by considering data flow, register conflicts, and side effect of code transformation? Is there a clean set of rules to

determine the impact of processor dynamic events such as stalls from program and data cache misses, and memory bank conflicts?

- Can STI for ILP be used for processors based on a superscalar architecture? Unlike VLIW/EPIC architectures, instructions are scheduled at run-time by the hardware in superscalar processors. What kind of run-time analysis is necessary and how do we use that information to drive the code transformation?

- Can STI for ILP be used on chip multiprocessors (CMP)? Though STI for ILP mainly targets VLIW uniprocessor platforms, it can be used for multiprocessors because there could a situation where number of available threads is greater than number of processors. In such case, integration of threads can help with using available processors more efficiently. In addition to increasing ILP of threads, locality of data cache can be leveraged by integrating threads which use same data. Further investigation is necessary to evaluate usefulness of this approach and define the method for determining which threads and how to integrate.

- Can STI for ILP be used with a dynamic scheduler in the operating system? Integrated threads can be invoked based on the dynamic and smart decision of the scheduler. Compared with the static approach, this 'Smart RTOS' approach offers some benefits such as exploiting parallelism across multiple applications, flexible execution of threads depending on dynamic decisions, and adaptation for changing workloads [86].

- Can STI be used for tolerating long memory latency? If there is a memory operation with long latency (possibly to multiple levels of memory hierarchy) and the architecture supports a non-blocking request, the idle time waiting for the response from the memory can be reclaimed by executing useful work by STI. If the idle time is fragmented, Asynchronous Software Thread Integration (ASTI) can be used because it allows independent and asynchronous progress for the primary and secondary threads [65].

7.3 Publications

- Won So and Alexander G. Dean. Reaching fast code faster: Using modeling for efficient software thread integration on a VLIW DSP. In *Proceedings of the 2006 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '06)*, pages 13–23. ACM Press, Oct. 2006

- Won So and Alexander G. Dean. Complementing software pipelining with software thread integration. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*, pages 137–146. ACM Press, June 2005.

- Won So and Alexander G. Dean. Procedure cloning and integration for converting parallelism from coarse to fine grain. In *Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architecture (INTERACT '03)*, pages 27–36. IEEE Computer Society, Feb. 2003.

7.4 Acknowledgements

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

Bibliography

[1] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure.

[2] A. Aiken and A. Nicolau. Loop quantization: an analysis and algorithm. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[3] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*, pages 221–235. Springer-Verlag, 1988.

[4] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO 31)*, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[5] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[6] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[7] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88)*, pages 241–249, New York, NY, USA, 1988. ACM Press.

[8] V. H. Allen, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the 25th annual international symposium on Microarchitecture (MICRO 25)*, pages 72–80, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[9] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '04)*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[10] P. Banerjee, J. Chandy, M. Gupta, I. Hodges, E.W., J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, 1995.

[11] Berkeley Design Technology. VLIW architectures for DSP: A two-part lecture. http://www.bdti.com/articles/vliw_icspat99.pdf, 1999.

[12] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[13] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[14] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90)*, pages 53–65, New York, NY, USA, 1990. ACM Press.

[15] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[16] D. Callahan, K. D. Cooper, R. T. Hood, K. Kennedy, L. M. Torczon, and S. K. Warren. Parallel programming support in parascope. In *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 91–106, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[17] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of 29th Hawaii International Conference on System Sciences*, Jan. 1996.

[18] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–357. IEEE Computer Society, 1997.

[19] P. Carribault, A. Cohen, and W. Jalby. Deep jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005)*, pages 291 – 302, Sept. 2005.

[20] P. P. Chang, N. J. Warter, S. Mahlke, W. Y. Chen, and W. W. Hwu. Three superblock scheduling models for superscalar and superpipelined processors. Technical report, University of Illinois, Urbana, IL, Dec. 1991.

[21] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(3):18–27, 1981.

[22] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[23] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.

[24] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on

program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.

[25] A. G. Dean. *Software Thread Integration for Hardware to Software Migration*. PhD thesis, Carnegie Mellon University, Feb. 2000.

[26] A. G. Dean. Compiling for fine-grain concurrency: Planning and performing software thread integration. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 103. IEEE Computer Society, 2002.

[27] A. G. Dean and J. P. Shen. Techniques for software thread integration in real-time embedded systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 322–333, 1998.

[28] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI '95)*, pages 93–102, New York, NY, USA, 1995. ACM Press.

[29] J. Eyre and J. Bier. Carmel enables customizable DSP. *Microprocessor Report*, 12(17), Dec. 1998.

[30] P. Faraboschi, G. Brown, J. Fisher, and G. Desoli. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 203–213, June 2000.

[31] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[32] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):278–490, 1981.

[33] M. Franklin. *The multiscalar architecture*. PhD thesis, University of Wisconsin at Madison, 1993.

[34] J. Fridman and Z. Greenfield. The TigerSHARC DSP architecture. *Microprocessor Report*, 12(16), Dec. 1998.

[35] J. Fritts, W. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. In *SPIE Photonics West, Media Processors '99*, pages 2–13, Jan. 1999.

[36] E. Gansner, E. Koutsofios, and S. North. *Drawing graphs with dot*, Jan. 2006.

[37] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proceedings of a conference on Functional programming languages and computer architecture*, pages 257–277. Springer-Verlag, 1987.

[38] P. Glaskowsky. MAP1000 unfolds at Equator. *Microprocessor Report*, 12(16), Dec. 1998.

[39] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303. ACM Press, 2002.

[40] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiciak. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proceedings of the 3rd Workshop on Media and Stream Processors*, Dec. 2001.

[41] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[42] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[44] T. R. Halfhill. StarCore reveals it's first DSP. *Microprocessor Report*, 13(6):13–16, May 1999.

[45] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.

[46] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[47] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transfomation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, 1994.

[48] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural analysis for parallelization. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC '95)*, pages 61–80. Springer-Verlag, 1996.

[49] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28)*, pages 158–168. IEEE Computer Society Press, 1995.

[50] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, page 266, Washington, DC, USA, 1998. IEEE Computer Society.

[51] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00)*, pages 113–123, New York, NY, USA, 2000. ACM Press.

[52] F. Homewood and P. Faraboschi. ST200: A VLIW architecture for media-oriented applications. In *Microprocessor Forum*, Oct. 2000.

[53] J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for TriMedia. *Journal of Instruction-Level Parallelism*, 1, 1999.

[54] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93)*, pages 258–267. ACM Press, 1993.

[55] H. C. Hunter, E. M. Nystrom, and W. W. Hwu. Applying scalable interprocedural pointer analysis to embedded applications. In *Proceedings of the 2004 Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES '04)*, 2004.

[56] W. W. Hwu and P. P. Chang. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (PLDI '89)*, pages 246–257, New York, NY, USA, 1989. ACM Press.

[57] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[58] Intel Corp. *Intel IA-64 Architecture Software Developer's Manual, Vols. I-IV, Rev. 1.1*, July 2000.

[59] Intel Corp. *Intel IA-64 Architecture Software Developer's Manual Specification Update*, Aug. 2001.

[60] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES '03)*, San Diego, CA, June 2003.

[61] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[62] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[63] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[64] S. Kittitornkun and Y. H. Hu. Programmable digital signal processor (PDSP): A survey. http://mcl.ce.kmitl.ac.th/~kksurin/papers/PDSP.pdf, 2003.

[65] N. J. Kumar, S. Shivshankar, and A. G. Dean. Asynchronous software thread integration for efficient software implementations of embedded communication protocol controllers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 37–46. ACM Press, 2004.

[66] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI '88)*, pages 318–328. ACM Press, 1988.

[67] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear analysis and optimization of stream programs. In *Proceedings of the ACM SIGPLAN 2003 conference on*

Programming language design and implementation (PLDI '03), San Diego, CA, June 2003.

[68] D. M. Lavery and W. W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29)*, pages 126–137. IEEE Computer Society, 1996.

[69] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*, pages 330–335. IEEE Computer Society, 1997.

[70] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design (ICCAD '99)*, pages 253–256, Piscataway, NJ, USA, 1999. IEEE Press.

[71] 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. *SIGMICRO Newsletter*, 23(1-2):45–54, 1992.

[72] K. S. McKinley, J. E. B. Moss, S. K. Singhai, G. E. Weaver, and C. C. Weems. Compiling for heterogeneous system: A survey and an approach. Technical Report UM-CS-1995-059, Dept. of Computer Science, Univ. of Massachusetts, Amherst, MA, Sept. 1995.

[73] D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall PTR, 2001.

[74] M. Narayanan and K. A. Yelick. Generating permutation instructions from a high-level description. In *Proceedings of the 6th Workshop on Media and Streaming Processors*, 2004.

[75] A. Nene, S. Talla, B. Goldberg, and R. Rabbah. *Trimaran - an infrastructure for compiler research in instruction-level parallelism - user manual*. New York University, 1998.

[76] C. J. Newburn. *Exploiting Multi-Grained Parallelism for Multiple-Instruction Stream Architectures*. PhD thesis, Carnegie Mellon University, Nov. 1997.

[77] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August. From sequential programs to concurrent threads. *IEEE Computer Architecture Letter*, 5(1):2, 2006.

[78] E. Ozer, T. M. Conte, and S. Sharma. Weld: A multithreading technique towards latency-tolerant VLIW processors. In *Proceedings of the 8th International Conference on High Performance Computing (HiPC '01)*, pages 192–203, London, UK, 2001. Springer-Verlag.

[79] S. Pillai and M. F. Jacome. Compiler-directed ILP extraction for clustered VLIW/EPIC machines: Predication, speculation and modulo scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, page 10422, Washington, DC, USA, 2003. IEEE Computer Society.

[80] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Int. J. High Speed Comput.*, 1(1):45–72, 1989.

[81] T. A. Proebsting and S. A. Watterson. Filter fusion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–130. ACM Press, 1996.

[82] Y. Qian, S. Carr, and P. Sweany. Loop fusion for clustered VLIW architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems (LCTES/SCOPES '02)*, pages 112–119. ACM Press, 2002.

[83] Y. Qian, S. Carr, and P. H. Sweany. Optimizing loop performance for clustered VLIW architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280. IEEE Computer Society, 2002.

[84] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming (MICRO 14)*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[85] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proceedings of the 9th annual symposium on Computer Architecture (ISCA '82)*, pages 131–139, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[86] W. So. Software thread integration for converting TLP to ILP on VLIW/EPIC processors. Master's thesis, North Carolina State University, Oct. 2002.

[87] W. So and A. G. Dean. Procedure cloning and integration for converting parallelism from coarse to fine grain. In *Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures (INTERACT '03)*, pages 27–36. IEEE Computer Society, Feb. 2003.

[88] W. So and A. G. Dean. Complementing software pipelining with software thread integration. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '05)*. ACM Press, June 2005.

[89] W. So and A. G. Dean. Reaching fast code faster: Using modeling for efficient software thread integration on a VLIW DSP. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pages 13–23. ACM Press, Oct. 2006.

[90] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95)*, pages 414–425. ACM Press, 1995.

[91] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[92] M. G. Stoodley and C. G. Lee. Software pipelining loops with conditional branches. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29)*, pages 262–273. IEEE Computer Society, 1996.

[93] E. Stotzer and E. Leiss. Modulo scheduling for the TMS320C6x VLIW DSP architecture. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*, pages 28–34. ACM Press, 1999.

[94] streamit@cag.lcs.mit.edu. *StreamIt Cookbook*, Oct. 2003.

[95] streamit@cag.lcs.mit.edu. *StreamIt Language Specification Version 2.0*, Aug. 2003.

[96] B. Su, S. Ding, J. Wang, and J. Xia. GURPR – a method for global software pipelining. In *Proceedings of the 20th annual workshop on Microprogramming (MICRO 20)*, pages 88–96. ACM Press, 1987.

[97] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[98] Texas Instruments. *Code Composer Studio User's Guide (Rev. B)*, Mar. 2000.

[99] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, Sept. 2000.

[100] Texas Instruments. *TMS320C64x Technical Overview*, Jan. 2001.

[101] Texas Instruments. *TMS320C64x DSP Library Programmer's Reference*, Apr. 2002.

[102] Texas Instruments. *TMS320C64x Image/Video Processing Library Programmer's Reference*, Apr. 2002.

[103] Texas Instruments. *TMS320C6000 DSP Peripherals Overview Reference Guide (Rev. G)*, Sept. 2004.

[104] Texas Instruments. *TMS320C6000 Optimizing Compiler User's Guide (Rev. L)*, Mar. 2004.

[105] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-620, Massachusetts Institute of Technology, Cambridge, MA, Aug. 2001.

[106] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[107] W. Thies, M. Karczmarek, M. Gordon, D. Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications. MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, Dec. 2001.

[108] J. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[109] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95)*, pages 392–403, New York, NY, USA, 1995. ACM Press.

[110] J. Turley and H. Hakkarainen. TI's new 'C6x DSP screams at 1,600 MIPS. *Microprocessor Report*, 11(2):14–17, Feb. 1997.

[111] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS-IV)*, pages 176–188, New York, NY, USA, 1991. ACM Press.

[112] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, 1992. ACM and IEEE.

[113] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93)*, pages 290–299, New York, NY, USA, 1993. ACM Press.

[114] N. J. Warter-Perez and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proceedings of the 28th annual international symposium on Microarchitecture (MICRO 28)*, pages 111–119, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[115] T. Way, B. Breech, and L. Pollock. Demand-driven inlining heuristics in a region-based optimizing compiler for ILP architectures. In *Proceedings of the 2001 IASTED International Conference on Parallel and Distributed Computing and systems (PDCS '01)*, pages 90–95, Anaheim, CA, USA, Aug. 2001.

[116] T. Way and L. Pollock. Using path spectra to direct function cloning. In *Workshop on Profile and Feedback-Directed Compilation*, pages 40–47, Oct. 1998.

[117] A. Wolfe and J. 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 (ASPLOS-IV)*, pages 2–14. ACM Press, 1991.

[118] H. Zhou and T. M. Conte. Code size efficiency in global scheduling for ILP processors. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT '02)*, page 79, Washington, DC, USA, 2002. IEEE Computer Society.

[119] V. Zivojnovic, C. Schlager, and H. Meyr. DSPStone: A DSP-oriented benchmarking methodology. In *International Conference on Signal Processing*, 1995.