Proceedings

Workshop on Interaction between Compilers and Computer Architectures

February 3, 2002
Cambridge, MA

In conjunction with

Eighth International Symposium on

High-performance Computer Architecture (HPCA-8)

The Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-6)

February 3, 2001 Cambridge, MA

In conjunction with
8th International Symposium on
High-performance Computer Architecture (HPCA-8)

Sponsored by
IEEE Computer Society
Technical Committee on Computer Architecture

Program Committee

Chair: Gyungho Lee, *Iowa State Univ.*

David August, Princeton *Univ.*

Todd Austin, *Univ. of Michigan*

Doug Burger, *Univ. of Texas-Austin*

Kemal Ebcioglu, *IBM*

Antonio Gonzalez, *Universitat Politecnica de Catalunya, Spain*

Lizy John, *Univ. of Texas-Austin*

Zhiyuan Li, *Purdue Univ.*

Eric Rotenberg, *North Carolina State Univ.*

Andre Seznec, *IRISA, France*

INTERACT-6: The Sixth Annual Workshop on Interaction between Compilers and Computers Architectures

Advance Program

I. Instruction Scheduling (9:00 am ~ 10:00pm)

Chair: G. Lee

Compiling for Fine-Grain Concurrency: Planning and Performing Software Thread Integration
Alexander G. Dean
Department of ECE, North Carolina State University

Dynamically Scheduling VLIW Instructions with Dependency Information
Sunghyun Jee, Chonan College in Foreign Studies
Kannappan Palaniappan, Department of CSCE, University of Missouri – Columbia

- Coffee Break (10:00 ~ 10:30am)

II. Simulation and Profiling (10:30 ~ 12:00am)

Chair: A. Dean

Accuracy of Profile Maintenance in Optimizing Compilers
Youfeng Wu
Microprocessor Research Labs, Intel

Mastering Startup Costs in Assembler-Based Compiled Instruction-Set Simulation
Ronan Amicel, and Francois Bodin
IRISA / INRIA, France

On the Predictability of Program Behavior Using Different Input Data Sets
Wei Chung Hsu, Howard Chen, Pen Chung Yew
Department of Computer Science, University of Minnesota
Dong-Yuan Chen, Microprocessor Research Labs, Intel

- Lunch Break (12:00 ~ 2:00pm)

III. Data Access (2:00 ~ 3:00pm)

Chair: G. Lee

Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors

R. Dave Weldon, Steven S. Chang, Hong Wang, Gerolf Hoflehner,
Perry Wang, Dan Lavery and John Shen
Microarchitecture Research Labs, Intel

Efficient and Fast Data Allocation of On-chip Dual Memory Banks

Jeonghun Cho, Jinhwan Kim, and Yunheung Paek
Department of EECS, Korea Advanced Institute of Science & Technology

- Coffee Break (3:00 ~ 3:30pm)

IV. Code Size (3:30 ~ 5:00pm)

Chair: W. Hsu

Code Size Efficiency in Global Scheduling for ILP Processors

Huiyang Zhou, and Thomas M. Conte
Department of ECE, North Carolina State University

Code Compression by Register Operand Dependency

Kelvin Lin, Jean Jyh-Jiun Shann, and Chung-Ping Chung
Department of CSIE, National Chiao Tung University

Code Cache Management Schemes for Dynamic Optimizers

Kim Hazelwood, and Michael D. Smith
Division of Engineering and Applied Sciences, Harvard University

Compiling for Fine-Grain Concurrency: Planning and Performing Software Thread Integration

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

Abstract

Embedded systems require control of many concurrent real-time activities, leading to system designs which feature multiple hardware peripherals with each providing a specific, dedicated service. These peripherals increase system size, cost, weight, power and design time. ***Software thread integration*** *(STI) provides low-cost thread concurrency on general-purpose processors by automatically interleaving multiple (potentially real-time) threads of control into one. This simplifies hardware to software migration (which eliminates dedicated hardware) and can help embedded system designers meet design constraints such as size, weight, power and cost.*

This paper introduces ***automated methods for planning and performing*** *the code transformations needed for integration of functions with* ***more sophisticated control flows*** *than in previous work. We demonstrate the methods by using* ***Thrint****, our post-pass thread-integrating compiler, to automatically integrate multiple threads for a sample real-time embedded system with fine-grain concurrency. Previous work in thread integration required users to manually integrate loops; this is now performed automatically. The sample application generates an NTSC monochrome video signal (sending out a stream of pixels to a video DAC) with STI to replace a video refresh controller IC. Using Thrint reduces integration time from days to minutes and reclaims up to 99% of the system's fine-grain idle time.*

1. Introduction

Embedded systems have multiple concurrent activities which must meet their deadlines or else the system will fail. These activities are usually implemented in hardware to guarantee they occur on time, as most microprocessors suffer when trying to perform multiple threads concurrently at a fine grain while meeting deadlines. Adding this hardware complicates system design whether added as external ICs or as modules on a microcontroller or system-on-chip. External components increase system size, weight, power, parts cost and design time. Integrated hardware peripherals increase design time and also fracture the chipmaker's market (which leads to increased cost through reduced volumes). In the end, both internal and external hardware solutions increase costs.

1.1 Hardware to Software Migration Challenges

These costs have led to many efforts to implement the concurrent activities in software in order to ride the wave of falling compute costs described by Moore's law. There are two difficulties with making generic microprocessors adept at executing multiple concurrent threads.

First, the processor must switch easily among contexts, saving and restoring registers with each switch. There are many techniques (register banks and windows, coarse- and fine-grained multithreading, simultaneous multithreading, multiprocessing) to allow quick switches [22][33][31][32][7][30][6]. Some of these techniques are available in embedded processors, though not all.

Second, the processor must execute the right instructions from the right thread at the right time. This is the crux of the problem. The general solution is to divide threads into coarse- and fine-grain pieces. Each coarse piece is made of concatenated fine-grain pieces. Scheduling the fine-grain pieces is done statically (at compile time) and involves executing padding instructions to generate a given time delay. Nops are typically used. Although some coders have painstakingly managed to inject by hand instructions which perform use-

ful work for another part of the program, the resulting programs are brittle and difficult to maintain. Furthermore, this approach is very poorly suited to systems which require frequent real-time activity with fine timing accuracy .

Despite these difficulties, there is an abundance of articles and application notes from makers of microcontrollers describing how to extract concurrency from their generic processors [3] [4] [13] [14] [19] [20] [23] [24]. These efforts primarily target two classes of applications: video signal generators and communication protocol controllers. This paper is in the first area. The second area is much more demanding, and we are currently extending our STI concepts to suppport it.

1.2 HSM with STI

We have developed and continue to enhance our compiler-based approach to providing fine-grain concurrency. We have developed a compiler *Thrint* which automatically creates an implicitly multithreaded function from two functions, one with real-time requirements on specific instructions. It implements many of the time-driven code transformations which we developed for integrating threads while maintaining control, data and timing correctness. With this technology an off-the-shelf uniprocessor can efficiently perform multiple concurrent functions without any special support for rapid context switching, scheduling or concurrency. This in turn makes hardware to software migration (HSM) viable.

Our past work developed concepts and methods not only for software thread integration (STI) but also how to use it for HSM, in which a real-time *guest thread* replaces the dedicated hardware and is integrated with one or more *host threads* from the application. We created a practical design procedure based upon idle time analysis of guest threads (to determine which threads have enough idle time to be worth integrating), temporal determinism of host threads (to determine which threads have enough timing determinism to be good frameworks for integration), methods for recognizing guest trigger events, methods for dispatching integrated threads, and techniques to predict overall performance of an integrated system. We derived methods for measuring the performance of the integrated software for peak and average cases as well [9][10][8][12].

In this paper we present the algorithms used to plan and then perform integration, using previously developed integration transformations as building blocks. We use two phases to enable the evaluation of a variety of integration plans using estimates of MIPS recovered, guest response latency and code expansion. This enables the compiler or the designer to select the best approach for integration. An NTSC video refresh application is used to demonstrate the automated use of the integration planning and execution methods presented here. We use our compiler *Thrint* for the integration; it now totals over 10,000 lines of C++ code in 20 modules. We examine an application and use *Thrint* to perform the video refresh work, creating efficient integrated functions.

This paper has the following organization. Section 2 gives an overview of how STI works. Section 3 introduces the planning and transformation algorithms and data structures used in *Thrint*. Section 4 examines the integration of a sample application. Section 5 summarizes the results and their broader implications.

2. Software Thread Integration Overview

Figure 1 presents an overview of how STI is used for HSM. A hardware function is replaced with software written by a programmer. This code consists of one or more guest threads (represented by the blue bar) with real-time requirements. When the threads are scheduled for execution on a sufficiently fast CPU, gaps will appear in the schedule of guest instructions, as illustrated by the white gaps in the blue bar. These gaps are pieces of idle time which can be reclaimed to perform useful host work. STI recovers fine-grain idle time efficiently and automatically.

STI uses a *control dependence graph* (CDG, a subset of the program dependence graph [1][15][17][18][26][27][29]]) to represent each procedure in a program. In this hierarchical graph (please see Figure 2), control dependences such as conditionals and loops are represented as non-leaf nodes, and assembly instructions are stored in leaf nodes. Conditional nesting is represented vertically while execution order is horizontal, so that an in-order left-to-right traversal matches the program's execution. The CDG is a good form for holding a program for STI because this structure simplifies analysis and transformation through its hierarchy. Program constructs such as loops and conditionals as well as single basic blocks are moved efficiently in a coarse-grain fashion, yet the transformations also provide fine-grain scheduling of instructions as needed.

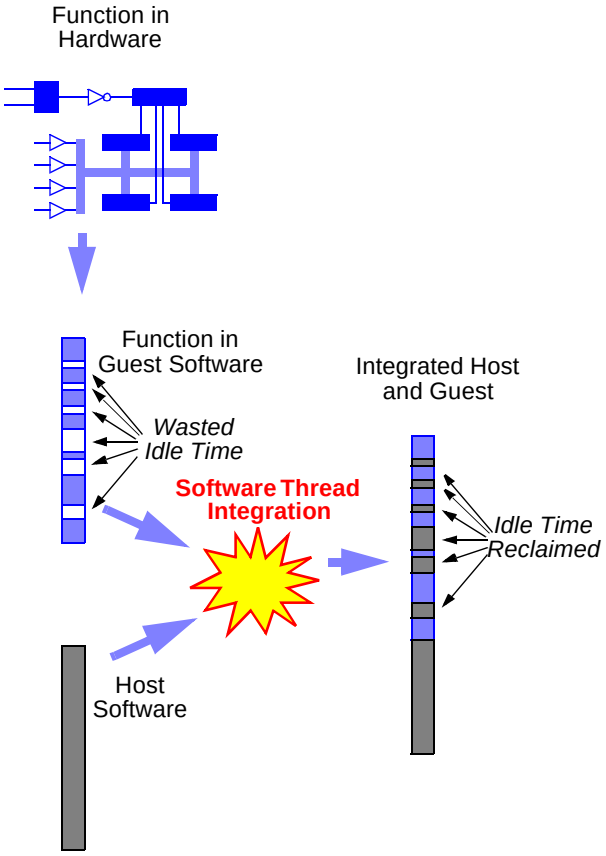


Figure 1. Overview of hardware to software migration with STI. Idle time is statically filled with useful work.

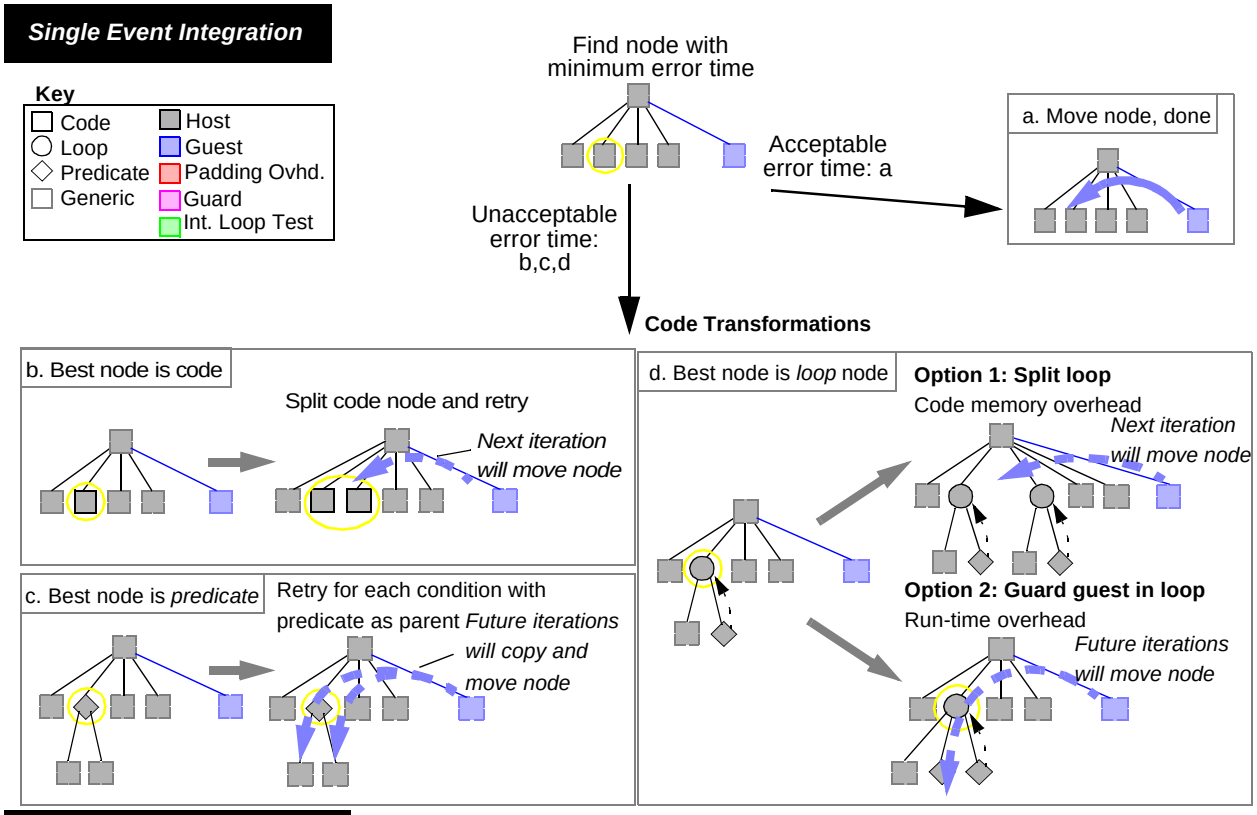
Using STI for HSM involves moving guest code into the correct position within the host code for execution at the correct time. The first stage in this code motion is called degenerate integration; the programmer manually appends the guest procedure code to the end of the host procedures. The resulting procedure is then automatically integrated by moving guest nodes left in the CDG to locations which correspond to the target time ranges. A tight target time range may fall completely within a host node, forcing movement down into that node or its subgraph. As shown in Figure 2, we have developed a set of CDG transformations [10][11][12] which can be applied repeatedly and hierarchically, enabling code motion into a variety of nested control structures. For example, moving a single guest event node into a host code node requires splitting the code node (a basic block). This is shown in the diagram as single event case b. Moving into a conditional (a predicate) requires guest replication into each case (single event case c). Moving into a loop requires loop splitting (single event case d1) or guarded execution (single event case d2) on a specific iteration. The transformations also support the integration of guest loops with host loops. Loop fusion, guarding and unrolling are used to match the host loop's work with the available idle time within one or more guest iterations. Most of these transformations are implemented in our thread integrating compiler *Thrint*.

STI automatically ensures semantic and timing correctness with its transformations. The variety of integration methods and decisions enable the STI tool to automatically optimize for execution speed or code size.

All control and data dependences must be observed to ensure semantic correctness. The CDG explicitly represents control dependences as graph structures; STI's code transformations modify the graph yet maintain these dependences. These transformations enable STI to interleave code from different threads, which is the key to reclaiming idle time efficiently. STI only needs to handle false data dependences when integrating threads; no other data dependency issues arise because each individual thread remains in order. Assembly code contains many false data dependences because of register reuse, so STI automatically reallocates registers to remove this constraint and make code motion easier.

All real-time dependences must be observed to ensure timing correctness. Each RT guest instruction must be moved to execute within its target time range. STI automates this process. First the host and guest threads are statically analyzed for timing behavior [25][28], with best and worst cases predicted. Hardware and software both conspire to make this a difficult problem in the general case. However, we focus on applications without recursion or dynamic function calls, and processors without superscalar execution, virtual memory or variable latency instructions. We assume locked caches or fast on-chip memory and no pipelining.

For perspective, in 1999 81% of the 5.3 billion microprocessors sold were four- and eight-bit units (9% were 16-bit). These MCUs run applications which are not computationally intensive, and do not need more parallelism or faster clock rates. Instead they are constrained by other issues such as functionality, cost, power dissipation, design time and use of commercial off-the-shelf products.



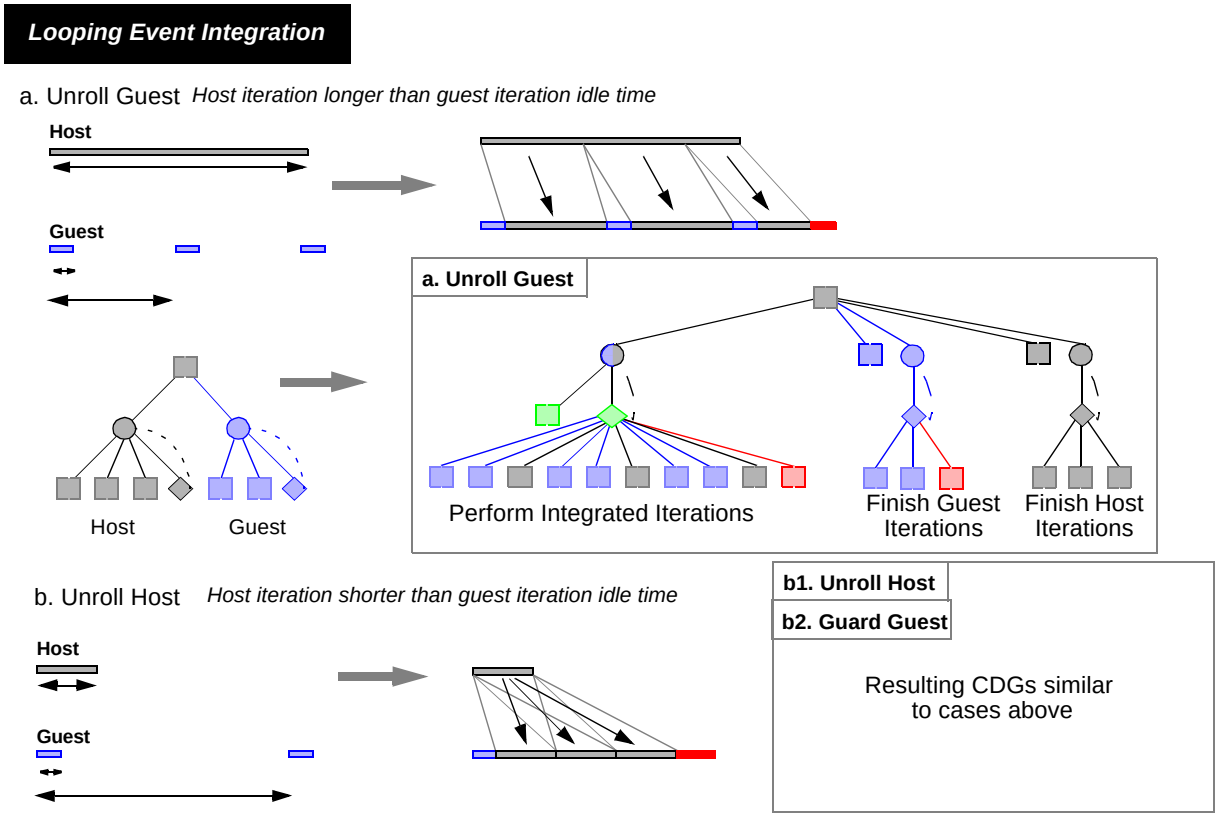


Figure 2. Summary of Single and Looping Event Code Transformations for STI

The fact that microarchitectural features such as superscalar execution and memory caches complicate the static timing analysis upon which STI relies is irrelevant for these applications, as they do not need the performance proided by these features. In fact, often these applications cannot afford the additional cost of such an enhanced processor.

During integration, timing directives (supplied by the programmer) guide integration. Timing jitter in the host thread is automatically reduced (using padding instructions) to meet guest requirements. The CDG's structure makes the timing analysis and integration straightforward.

STI produces code which is more efficient than context-switching or busy-waiting. The processor spends fewer cycles performing overhead work. The price is expanded code memory. STI may duplicate code, unroll and split loops and add guard instructions. It may also duplicate both host and guest threads. Memory may be cheap, but it is not free. The memory expansion can be reduced by trading off execution speed or timing accuracy. This flexibility allows the tailoring of STI transformations to a particular embedded system's constraints.

3. Planning and Performing Integration

Integration requires several stages of preparation. Time-critical guest code is identified based upon user directives, the host code's execution schedule is predicted for both best and worst cases, and temporally deterministic segments within the host are identified as targets for integration (as described in [11]). Next, integration planning takes place. Thrint plans the integration transformations needed to integrate the guest function with each of the temporally deterministic segments identified previously. The guest code is then integrated with one or more of these temporally deterministic segments.

The guest thread consists of nodes (code, loop, predicate), some of which may have timing requirements associated with them (code and loop). These are sub-thread timing requirements; thread-level timing requirements are dealt with elsewhere. We use timing directives to specify the target time (with a user-defined tolerance) for the start of the node's execution, as represented as a delay from the beginning of the integrated thread's execution. We call guest nodes with these timing requirements *explicitly specified guests* (or simply *explicit guests*); the other guest nodes are called *implicit guests*. Thread integration must place the explict guests in the host code based on the timing directives, while the implicit guests are merely constrained to be moved to ensure in-order execution between the explicit guests.

```
PROCEDURE DrawSprite INTO DrawSprite
   TOLERANCE 0
    BLOCK Video_Reset_Ptr AT 10
    LOOP Video_Loop PERIOD 40 ITERATION_COUNT 128
    BLOCK Video_Pix0 INTO_LOOP Video_Loop FIRST_AT 22
    BLOCK Video_Pix1 INTO_LOOP Video_Loop FIRST_AT 32
    BLOCK Video_Pix2 INTO_LOOP Video_Loop FIRST_AT 42
    BLOCK Video_Pix3 INTO_LOOP Video_Loop FIRST_AT 52
   BLOCK Video_End IMPLICIT
END
```

Figure 3. DrawSprite integration directives file specifies timing requirements for guest thread components

The integration requirements are defined in a text file as shown in Figure 3. They are loaded into a data structure which duplicates the CDG structure of the explicit guest nodes, as shown in Figure 4.

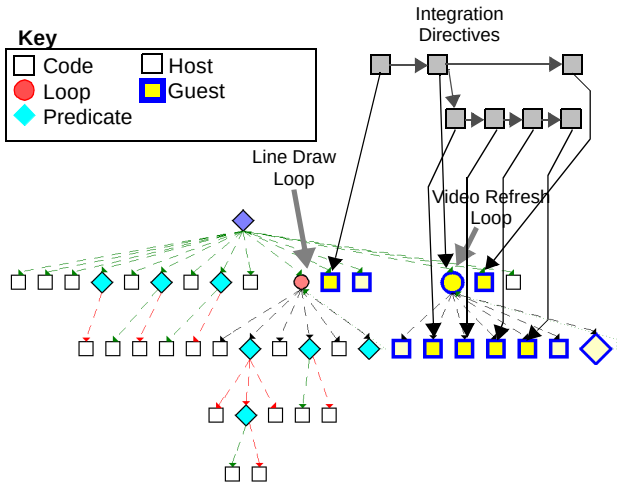


Figure 4. Initial DrawLine control dependence graph with integration directives data structure marking explicit guests

Each integration directive node holds a pointer to its explicit guest node, as well as a list of pointers to the implicit guests which precede and follow the explicit guest. Currently we limit explicit guests to be code or loop nodes which are at the first or second level of the CDG. We have found this to be adequate for a variety of applications. An implicit guest may be arbitrary code, provided that it is structured.

At this point integration planning begins, using the temporally deterministic segments [11] identified elsewhere (each of these is a contiguous subgraph of the host CDG).

The algorithm Plan_Integration (Figure 5) is used to identify which transformations are needed to integrate the guest code with the host segment. Plan_Integration creates an integration plan based on the integration directives data structure created previously, and then steps through each explicit guest within it.

```
IntegrationDirectives::Plan_Integration(host_segment) {
for each explicit guest in integration directives list
    if current guest is single guest event
        host_segment->get_first_node->find_host for current guest
    else // is looping guest event
        while guest loop iterations remain
            find time covered by these guest iterations
            if host loop execution overlap
                for each host loop executing during this time (in order)
                    if guest loop starts first
                        peel preceding iterations from guest loop
                        plan integration as multiple non-looping guests
                    else if host loop starts first
                        split preceding host loop iterations
                    if overlap time of host and guest loops is enough
                        plan to unroll guest or host loop
                        mark for fusion
                        if loop iterations remain
                            mark for clean-up loop copies
            else no host loops during guest loop
                peel all loop iterations
                plan their integration as multiple non-looping guests
}
```

Figure 5. Plan Integration finds hosts, planning loop transformations as needed to fuse loops

Integration for code guest nodes is handled by calling Find_Hosts (Figure 6). This algorithm identifies which node(s) will be executing during the guest's target time range, and determines which transformations (previously presented in Figure 2) are needed to ensure that if the guest is placed there, control-flow and timing requirements are met. This may involve determining where to split a loop or how many padding nops to use for balancing a predicate node.Figure 7 shows which target host nodes are identified for the example presented later in this paper.

Looping guest events require a more sophisticated approach, which is listed in Figure 5 (an example is graphically presented in Figure 8). The technique attempts to perform loop fusion if a guest and host loop overlap for multiple iterations. The goal of this loop fusion (as seen in Figure 2) is to match guest loop idle time with host loop body work through unrolling.

Guarded clean-up loops can be added following the fused loop body accomodate extra iterations or unknown loop counts. The portions of loops which do not overlap are handled differently depending upon type. Host loops are split to separate the overlapping and non-overlapping iterations. Guest loops have iterations peeled off, with each explicit guest in the unrolled iteration integrated as a non-looping guest node.

```
Node::Find_Hosts(target_time, guest_node_integration_plan) {
if this node finishes executing before target_time
    call Find_Hosts(target_time) on later sibling
else
    plan to pad node if needed to remove unacceptable jitter
    if target_time follows padded node
        // can execute guest immediately after this node
        guest_node_integration_plan->Add_Host(this, AFTER)
    else
        // descend into this node for better timing accuracy
        switch on this node's type
        CODE:
            guest_node_integration_plan->Add_Host(this, WITHIN)
        PREDICATE:
            // descend into condition subgraphs
            for each condition
                this->Get_First_Child(condition)->Find_Hosts(target_time,
                guest_node_integration_plan)
        LOOP:
            // descend into loop subgraph
            if guest_node is not in loop
                // plan to split host loop or guard guest within it
                this->Find_Hosts_In_Loop(target_time,
                guest_node_integration_plan)
            else
                // Plan_Loop_Integration() has already planned loop
                // fusion, so just locate guests in appropriate hosts
                // without additional loop transformations
                this->Get_First_Child()->Find_Hosts(target_time,
                guest_node_integration_plan)
}
```

Figure 6. Find_Hosts algorithm identifies host nodes and transformations to ensure guest begins execution within target_time range

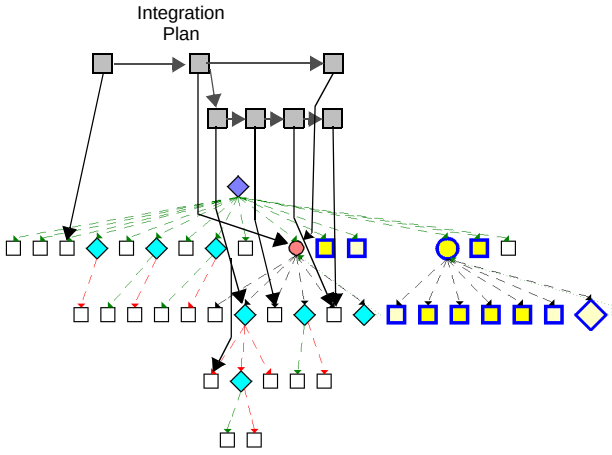


Figure 7. Integration plan data structure identifies locations in host code corresponding to guest target times

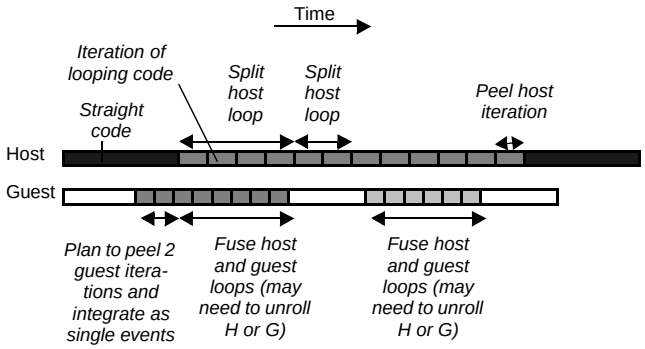


Figure 8. Diagram showing example of loop integration planning with fusion, splitting and peeling

This completes the planning for integration, allowing the evaluation of interesting evaluation plans to trade off code memory expansion for increased performance. One or more such integration plans may be selected for actual integration, as described in [12].

```
DeterministicCodeSegment::Integrate(integration_plan) {
for each top_level_guest
    if guest is loop
        for each host of top-level guest
            if current host is loop transform
                do loop transformation to prepare for fusion
            make guarded copy of guest loop after host loop if needed
        cur_guest = top_level_guest's first child
    else
        cur_guest = top_level_guest
    do
        pad previous nodes if needed  to cut start time jitter
        pad host node if needed  to cut host node's completion time jitter
        for each of cur_guest's hosts cur_host
            do any host loop transformations (e.g. splitting)
            update cur_host pointer based on previous guests, host
            splitting and padding
            insert these guest node clones at cur_host:
                cur_guest's previous implicit guest nodes
                cur_guest (explicit guest) node
                cur_guest's following implicit guest nodes
            advance cur_host
        if cur_guest within guest loop
            advance to next guest within guest loop
        else
            cur_guest = NULL
    while cur_guest
for each top_level_guest
    create and insert fused loop control tests
}
```

Figure 9. Integrate algorithm implements code transformations planned previously

Integration, presented in Figure 9, performs padding, loop splitting, unrolling and other transformations previously planned and then copies the guest nodes to the appropriate locations in the host code. Note that each explicit guest may be assigned multiple hosts, and each explicit guest may have multiple implicit guests.

4. STI for Video Application

Our previous work has developed concepts, code transformations and analytical methods for performing STI especially for HSM. Previous thread integration results reflect manually integrated code and a mix of manual and automatic analysis. In this paper we demonstrate automatic thread integration using our post-pass compiler *Thrint*, which implements automatic thread analysis, visualization, and integration by using techniques of control- and data-flow analysis, static timing analysis, code transformations and register reallocation. We first examine the application, then evaluate idle time within the guest thread and temporal determinacy within the host threads. We then analyze automatically integrated code for system efficiency and memory expansion.

4.1 Target System for Hardware to Software Migration

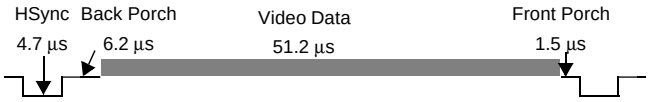


Figure 10. Video signal timing

To demonstrate the benefits of STI for HSM we use an NTSC video refresh controller application (for driving a CRT). We replace a video generator chip with a software version. The processor must generate an NTSC-compatible monochrome video signal [16], summarized in Figure 10. Although the beam scans 525 times per frame (in two interlaced passes (fields) per 33.3 ms frame), only 494 rows are visible and require video data, corresponding to 75.8% of the processor's time. There are additional features in a video signal (vertical sync, serration and equalization pulses) but these can be generated easily with standard methods (ISRs triggered by an on-chip timer) so we do not examine them in this work. The video data portion of the signal is the most demanding, as a pixel of video data must be generated every 100 ns (for 512 pixels per row). With a 100 MHz CPU this corresponds to ten clock cycles per pixel, which is very frequent and offers little time for context switching or scheduling.

We target a 32 bit scalar RISC processor running at 100 MHz with on-chip single-cycle memory access and instruction execution and no virtual memory. The

processor executes 100 million instructions per second. The hardware is structured as shown in Figure 11.

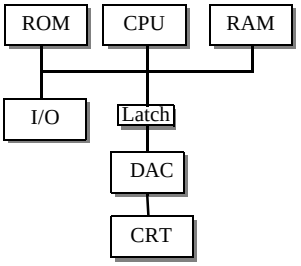


Figure 11. Hardware architecture of system lacks a video refresh controller

The system is designed to generate a monochrome 512x494 pixel image with eight bits per pixel. A digital-to-analog converter (DAC) converts the data byte to an analog voltage for the CRT. Note that this design

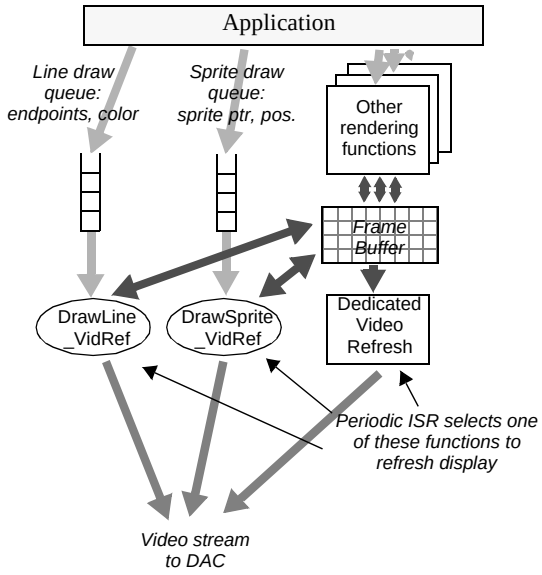


Figure 12. Video data flow overview

can easily be extended to provide color video to a CRT with RGB inputs by adding two more DACs and a look-up table (palette) memory.

We assume a periodic interrupt will trigger an ISR just before the beginning of the video data portion of each row, and that this ISR examines two queues which hold data to be used by two functions which have been integrated with display refresh code. The queues hold parameters for drawing lines or sprites and are fed by other functions in the application. The ISR selects one of the two integrated functions (if data is present in the queue) or else a dedicated busy-wait refresh function. The chosen thread then reads video data from the frame buffer in memory and sends it out to the CRT through the DACs.

The ISR and the queues are not implemented for this paper because they are straightforward to implement and analyze. Instead we focus on the integration and analysis of the refresh/render threads which are integrated by the compiler.

4.2 Experimental Method

Our compiler *Thrint* processes functions compiled for the Alpha instruction set architecture. Although it is not representative of most embedded systems, it was chosen to leverage the compiler *PCOM* from another tool suite (*Pedigree* [26]). The Alpha ISA is a clean load-store architecture with an ample register set and is a suitable target for this work. We assume a microarchitecture with easily predicted performance: scalar execution, single-cycle memory system or lockable cache, a predictable pipeline and no virtual memory. As explained in the introduction, the bulk of the applications targeted by this research neither need nor can afford the high throughputs provided by sophisticated and complex microarchitectures.

The guest (VidRef) and host functions (DrawLine [2] and DrawSprite) are written in C++ and used for initial degenerate integration (the guest function body is concatenated with the host function body, and automatic variables are copied). The new functions are compiled with gcc 2.7 with -O1 optimization. Basic block labels are added to the resulting assembly language functions to identify instructions with specific real-time requirements. The functions are then processed by *PCOM* into CDG-structured assembly language. These functions are analyzed and integrated by *Thrint*, which creates an output assembler file as well as visualization support files (e.g. Figure 14). Data symbol information is added to the assembler file (after having been deleted during processing) and then assembled. The object file is linked with an X-windows-based driver program to allow execution-based verification of program operation. Timing correctness is verified by static timing analysis in *Thrint* after all code transformations have been completed. As the target machine architecture is highly predictable (completely predictable pipeline, scalar instruction execution, single-cycle memory access), timing verification through execution or simulation is not performed.

4.3 Guest Thread

The video signal has events which must occur within tight time ranges to create a compatible video signal. As mentioned previously, we assume an interrupt service routine triggered by a programmable timer generates the signal transitions needed for the horizontal and vertical sync, front and back porches, and equalization and serration pulses. The video data is read out of a frame buffer and sent out through an 8 bit DAC by the previously mentioned VidRef function. When the software needed to perform these events is executed by the 100 MHz processor (without any context switching or scheduling overhead considered), the resulting idle time is distributed as shown in Figure 13.

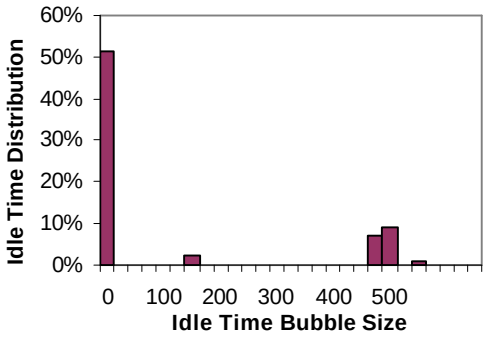


Figure 13. Idle time in video signal generation software is mostly fine-grain

The idle time in large bubbles (compared with the overhead of setting up a timer and performing two context switches (e.g. 30 cycles) is best recovered through context switching. The idle time in smaller bubbles is recovered through STI. Figure 13 shows that over half the processor's idle time for this video refresh application is in fine-grain pieces of idle time (four, seven and eight cycles), making this type of application a good fit for HSM using STI.

4.4 Host Threads

Figure 14 shows the control structure of DrawLine after the guest code has been appended and the file assembled. DrawLine takes two endpoints and a color code as arguments and scan converts the line into the frame buffer. Two conditionals (predicates) in the beginning of the function determine line direction, then the code determines increment values and finally a loop sets pixels in the frame buffer. The conditionals within the loop selectively update x or y counters and error variables. The guest code consists of pointer initialization code and then a loop which loads 32-bit words from memory and sends them out to the video display one byte at a time.

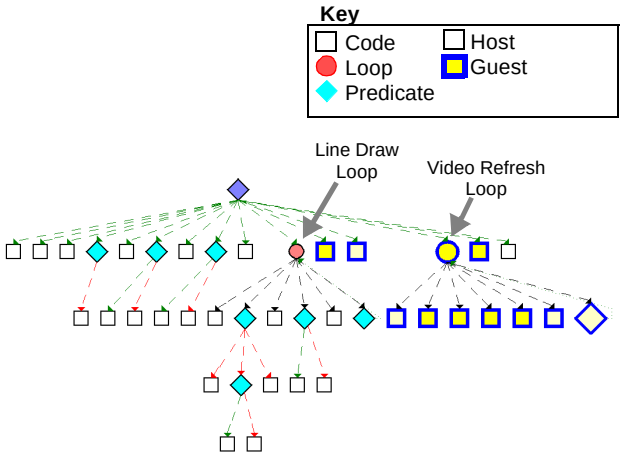


Figure 14. Initial DrawLine Control Dependence Graph

Figure 15 shows the control structure for DrawSprite with the guest code appended. DrawSprite takes a pointer to a sprite (a 16x16 pixel array) with a position and draws the sprite in the frame buffer. The code consists of a loop which iterates across sprite rows, and conditionals within that body which handle various position cases of the sprite. The guest code is the same as for DrawLine.

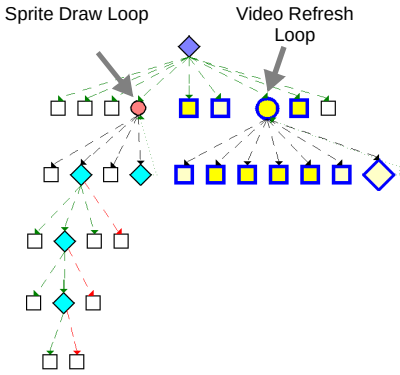


Figure 15. Initial DrawSprite Control Dependence Graph

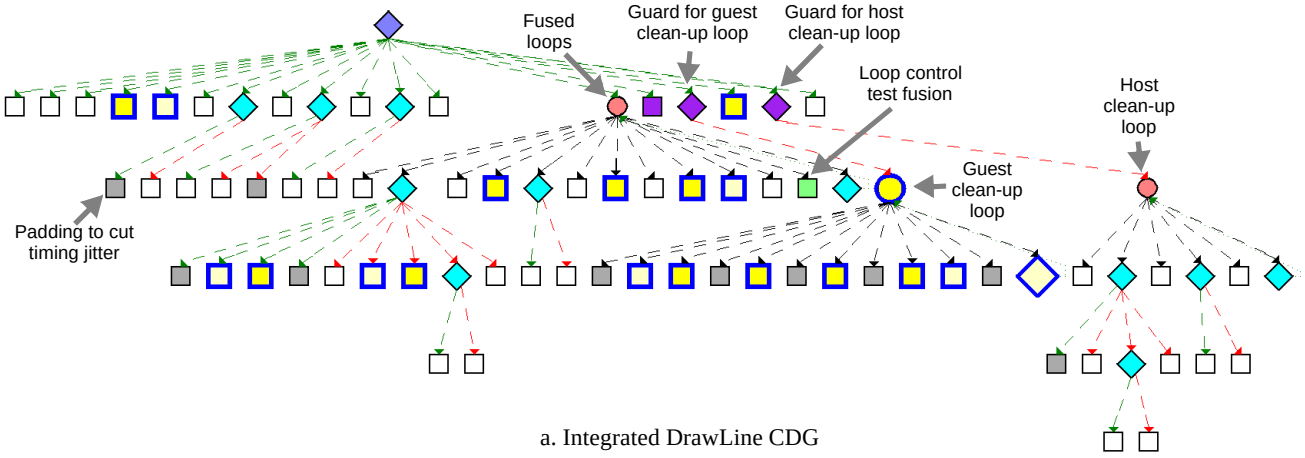
4.5 Integration Process

The code is prepared for integration by marking time-sensitive instructions in the guest thread assembly code (called *explicit guests*) with labels, and then specifying timing requirements for those basic blocks. Figure 3 presents the integration directives file used for integrating DrawSprite. A timing error tolerance of 0 cycles is specified, so *Thrint* pads away all timing jitter (leading to increased code size). Only explicit

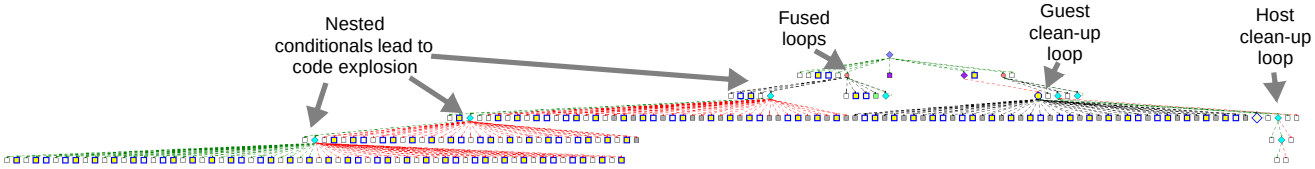
guests (nodes with specific timing requirements) need to be defined in this file; the intervening nodes are called *implicit guests* and are automatically handled. At this point *Thrint* is run to perform integration.

Figure 16 shows the CDGs of the two integrated functions. Figure 16.a shows that DrawLine's host loop is fused with the video refresh loop, and guest code is replicated into one host conditional. The loop control tests are fused with a logical AND to control loop execution. After the fused loop finishes, a guarded dedicated guest loop completes any remaining video refresh work. It is guarded to keep it from running if there is no more work. Following that loop is a guarded replica of the host loop to finish drawing long lines which were not completed in the fused loop.

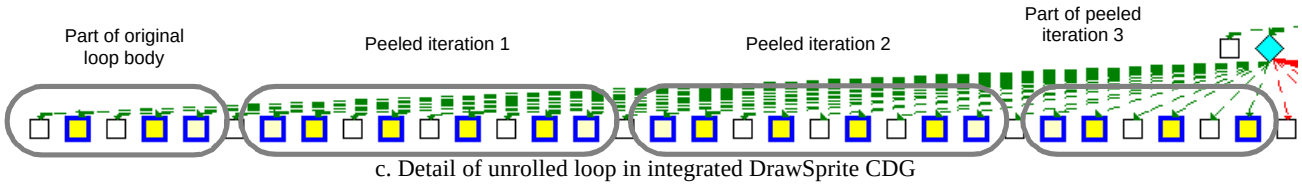
Figure 16.b shows DrawSprite integrated with VideoRefresh. The idle time in each iteration of the guest loop (27 cycles) is not long enough to hold a full host loop iteration (up to 107 cycles). *Thrint* unrolls the guest loop by ceil(107/27) = 4 times to fit the host within the idle time. Figure 16.c shows that part of the integrated unrolled loop has been copied into the taken case of a host predicate (conditional). The three nested levels of conditionals within DrawSprite lead to significant expansion because most of the guest code is replicated into these cases. Apart from this difference, the resulting code is similar in structure to the other example, with a dedicated guest clean-up loop followed by a host clean-up loop.



a. Integrated DrawLine CDG



b. Integrated DrawSprite CDG



c. Detail of unrolled loop in integrated DrawSprite CDG

Figure 16. Integrated CDGs

4.6 Code Expansion

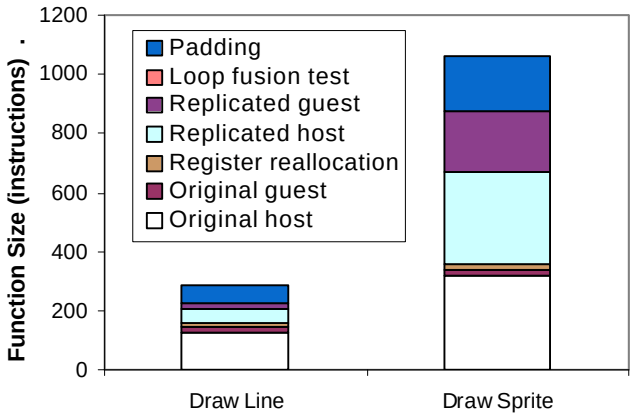


Figure 17. Code expansion for integrated threads

Figure 17 shows how STI affects function size for the examples used. DrawLine grows by 96% (from 144 to 282 instructions) and DrawSprite grows by 211% (from 341 to 1062 instructions). The bulk of the code expansion for DrawLine() (57 instructions) comes from padding to equalize timing variations among paths and to statically schedule the dedicated guest clean-up loop. Next are 47 instructions from splitting the host loop (marked "Replicated Host" in the graph). The guest requires some replication into conditionals, adding 18 instructions.

DrawSprite() grows first because its 312-instruction host loop is split to allow integration with its first 11 iterations. Next, the guest replication adds 206 instructions because the guest loop is unrolled three times, and much of that code must be replicated into each path of the three-conditional deep host loop. In addition, 187 padding instructions are added to statically schedule the guest code and reduce timing variations in the host code. Clearly the combination of deep conditionals and loop unrolling leads to code explosion; it would be logical to examine the DrawSprite function and replace the conditionals (which allow sprites to be drawn at positions unaligned with word boundaries) with computation.

This code expansion only applies to functions which are integrated. In a typical application there would only be a few, so the overall impact on code size would be slight.

4.7 Performance

Using STI to reclaim the idle time within the video refresh portion enables the system to perform more useful work such as line or sprite drawing. Figure 18 shows the system behavior as a function of line or sprite drawing performed per video row. The vertical axis presents the fraction of CPU time remaining after performing both video refresh and either line or sprite drawing. We examine the cases of a discrete video refresh (with all idle time filled with nops) and the integrated versions described previously. Performing the video refresh without line or sprite drawing immediately reduces free time to 18% for both implementations. However, as graphics work is added, the integrated version is able to recover idle time to perform useful work, raising drawing throughput for lines by 225% and sprites by 290% (X-axis intercepts).

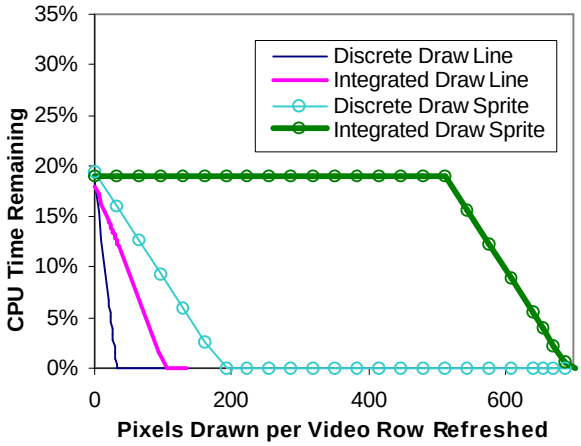


Figure 18. Integrated threads reclaim CPU for useful work, improving performance

The initially horizontal slope of the integrated DrawSprite shows that it recovers all available idle time. The non-horizontal slope of DrawLine shows that the overhead of integration consumes a significant amount of idle time.

5. Conclusions and Current Work

We have developed an automated solution to the hardware-to-software migration challenge called software thread integration. Our compiler *Thrint* automatically integrates multiple threads into a single implicitly multithreaded flow of control which executes on a standard uniprocessors without special support for scheduling or fast context switching. We have also developed design methods for using software thread integration to perform hardware to software migration quickly and efficiently.

In this paper we present the data structures and algorithms needed to plan and perform software thread

integration, using the transformations developed in previous work. These methods have been implemented in our research compiler Thrint. We demonstrate the results of automatic integration of a sample application with fine-grain concurrency and analyze the resulting code expansion.

We use *Thrint* and STI to replace a video refresh controller with a software implementation. We reduce integration time from days to minutes, paying a minor penalty in memory size while reclaiming large amounts of idle time.

6. References

[1] V.H. Allan, J. Janardhan, R.M. Lee and M. Srinivas: Enhanced Region Scheduling on a Program Dependence Graph, *Proceedings of the 25th International Symposium and Workshop on Microarchitecture (MICRO-25)*, Portland, OR, Dec. 1-4, 1992

[2] J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal*, 4(1), 1965, pp. 25-30

[3] Tony Breslin, "68HC05K0 Infra-Red Remote Control," Motorola Semiconductor Application Note AN463, 1997

[4] Dave Bursky, "Speedy 8-Bit Microcontroller Crafts Virtual Peripherals," *Electronic Design*, August 4, 1997

[5] G.J. Chaitin, "Register Allocation & Spilling via Graph Coloring," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98-105, June 1982

[6] D. Culler, "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1991

[7] Y. Chou, D. Siewiorek, J. Shen. "A Realistic Study on Multithreaded Superscalar Processor Design" Europar '97, Passau, Germany, August 1997

[8] Alexander G. Dean and Richard R. Grzybowski, "A High-Temperature Embedded Network Interface using Software Thread Integration," Second Workshop on Compiler and Architectural Support for Embedded Systems, Washington, DC, October 1-3 1999

[9] Alexander G. Dean and John Paul Shen, "Hardware to Software Migration with Real-Time Thread Integration," Proceedings of the 24th EUROMICRO Conference, Västerås, Sweden, August 25-27 1998, pp. 243-252.

[10] Alexander G. Dean and John Paul Shen, "Techniques for Software Thread Integration in Real-Time Embedded Systems," Proceedings of the 19th Symposium on Real-Time Systems, Madrid, Spain December 2-4 1998, pp. 322-333

[11] Alexander G. Dean and John Paul Shen, "System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection," Proceedings of the 20th Symposium on Real-Time Systems, Scottsdale, Arizona, December 1-3 1999, pp. 234-245

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

[13] Martin Embacher, "Replacing Dedicated Protocol Controllers with Code Efficient and Configurable Microcontrollers -- Low Speed CAN Network Applications," National Semiconductor Application Note 1048, August 1996

[14] Scott Fink, "Hardware/Software Trade-offs in Microcontroller-based Systems," Application Note, Microchip Inc., 1997

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

[16] B. Grob, **Basic Color Television Principles and Servicing,** McGraw Hill, 1975

[17] Rajiv Gupta and Mary Lou Soffa: Region Scheduling, *Proceedings of the Second International Conference on Supercomputing*, pp. 141-148, 1987

[18] Rajiv Gupta and Madalene Spezialetti: Busy-Idle Profiles and Compact Task Graphs: Compile-time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks, *15th IEEE Real Time Systems Symposium*, pp. 86-96, 1994

[19] Scott George, "HC05 Software-Driven Asynchronous Serial Communication Techniques Using the MC68HC705J1A," Motorola Semiconductor Application Note AN1240

[20] Thomas F. Herbert, "Integrating a Soft Modem," *Embedded Systems Programming*, 12(3), March 1999, pp. 62-74

[21] HD61830/HD61830B LCDC (LCD Timing Controller) Data Sheet. Hitachi, Inc.

[22] Yasuo Hidaka, Hanpei Koike and Hidehiko Tanaka, "Multiple Threads in Cyclic Register Windows," Proceedings of the 20th International Symposium on Computer Architecture, San Diego, CA, May 1993

[23] Stephen Holland, "Low-Cost Software Bell-202 Modem," *Circuit Cellar*, June 1999, #107, pp. 12-19

[24] Robert Lacoste, "PIC'Spectrum Audio Spectrum Analyzer," *Circuit Cellar*, September 1998, #98, pp. 24-31

[25] Sharad Malik, Margaret Martonosi and Yau-Tsun Steven Li: Static Timing Analysis of Embedded Software, *ACM Design Automation Conference*, June 1997.

[26] Chris J. Newburn, Derek B. Noonburg and John P. Shen: A PDG-Based Tool and Its Use in Analyzing Program Control Dependences, *International Conference on Parallel Architectures and Compilation Techniques*, 1994

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

[28] P. Puschner and C. Koza: Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):160-176, September 1989

[29] Madalene Spezialetti and Rajiv Gupta: Timed Perturbation Analysis: An Approach for Non-Intrusive Monitoring of Real-Time Computations, *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994

[30] R. Thekkath and S.J. Eggers, "The Effectiveness of Multiple Hardware Contexts," Proc. 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1994

[31] Toshiba America Electronic Components, Inc., "Clock Speeds Double With 20MHz Neuron Chips," Press Release, September 28 1997

[32] Toshiba America Electronic Components, Inc., "TMPN3120A20M, TMPN3120A20U Data Sheet," August 1998

[33] Carl A. Waldspurger and William E. Weihl, "Register Relocation: Flexible Contexts for Multithreading," Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, May 1993

Dynamically Scheduling VLIW Instructions with Dependency Information

Sunghyun Jee
Chonan College in Foreign Studies
Chonan, Chungnam, South Korea
jees@missouri.edu

Kannappan Palaniappan
University of Missouri
Missouri, Columbia, U.S.A.
palani@cecs.missouri.edu

Abstract

This paper proposes balancing scheduling effort more evenly between the compiler and the processor, by introducing dynamically scheduled Very Long Instruction Word (VLIW) instructions. Dynamically Instruction Scheduled VLIW (DISVLIW) processor is aimed specifically at dynamic scheduling VLIW instructions with dependency information. The DISVLIW processor dynamically schedules each instruction within long instructions using functional unit and dynamic scheduler pairs. Every dynamic scheduler dynamically checks for data dependencies and resource collisions while scheduling each instruction. This scheduling is especially effective in applications containing loops. We simulate the architecture and show that the DISVLIW processor performs significantly better than the VLIW processor for a wide range of cache sizes and across various numerical benchmark applications.

1. Introduction

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel using a combination of compiler and hardware techniques. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of ILP by executing long instructions composed of multiple instructions. The VLIW processor has performance bottlenecks due to static instruction scheduling and the unoptimized large object code containing a number of NOPs (No OPerations) and LNOPs (Long NOPs), where the LNOP means a long instruction that has only NOPs [20~22]. Superscalar VLIW (SVLIW) is the improving style of VLIW processor design that tries to execute object code constructed by removing all LNOPs from VLIW code [14,15,21,22]. The SVLIW processor also has a performance limitation similar to the VLIW processor due to static scheduling. By making use of powerful features to generate high-performance code, the IA-64 architecture allows the compiler to exploit high ILP using Explicit Parallel Instruction Computing (EPIC) [23,24]. The IA-64 is a statically scheduled processor architecture where the compiler is responsible for efficiently exploiting the available ILP and keeps the executions busy [24]. Instead of the merits, the IA-64 processor has a performance limitation due to static instruction scheduling. In order to overcome current performance bottlenecks in modern architectures, a processor architecture that satisfies the following criteria is required: (1) balanced scheduling effort between compile-time and run-time, (2) dynamic instruction scheduling, and (3) reducing the size of object code.

This paper presents a new ILP processor architecture called Dynamically Instruction Scheduled VLIW (DISVLIW) that achieves these goals. The DISVLIW instruction format is augmented to allow dependent bit vectors to be placed in the same VLIW word. Dependent bit vectors are added to the instruction format to enable synchronization between prior and subsequent instructions. To schedule instructions dynamically, the DISVLIW processor uses functional unit and dynamic scheduler pairs. Every dynamic scheduler decides to issue the next instruction to the associated functional unit, or to stall the functional unit due to possible resource collisions or data dependencies among instructions per every cycle. Such features can reduce the total number of execution cycles of the DISVLIW processor better than those of the VLIW or the SVLIW processor that compulsorily schedules long instructions. The DISVLIW processor is reminiscent of the CDC-6600 Scoreboard, an early dynamically scheduled processor architecture [22]. A different with the CDC-6600 is that the compiler conveys more explicit information for managing the scoreboard, in the form of the dependence bit vectors. Besides, even though the superscalar processor is an effective way of exploiting ILP, this superscalar processor architecture requires complex devices and the impact of such complexity on the design cost and clock cycle time can be severe [20,21]. Consequently, the superscalar processor will not be evaluated in this paper.

The rest of the paper is organized as follows. Section 2 compares issue slots and instruction pipelines of various ILP processors, Section 3 introduces the DISVLIW

processor architecture and instruction pipeline, in Section 4 we evaluate a performance of the DISVLIW processor, and conclusion follows in Section 5.

2. Instruction level parallelism

Figure 1 shows issue slots and execution images of the VLIW, the SVLIW, and the DISVLIW processors. The processors execute their own object code generated from given data dependency graph. In the data dependency graph, a node represents an instruction and a directed edge is annotated with data dependencies and resource collisions between instructions. We assume that every processor has three untyped functional units that can execute any instruction and a long instruction has three instructions. Figure 1 illustrates issue slots of each object code using rectangles repeated in the horizontal direction to represent consecutive clock cycles. Squares placed vertically in each rectangle represent the per cycle utilization instruction issue slots, where a rectangle and a square mean a long instruction and an instruction. An instruction is executed in the following four stages: F (Fetch), D (Decode), EX (EXecute), and WB (Write Back).

Figure 1(b-1) shows an execution image for the VLIW processor to execute VLIW code. The VLIW code contains a number of LNOPs and NOPs in order to solve data dependencies and resource collisions between long instructions as shown in Figure 1(a-1). During execution, the VLIW processor does not allow the next long instruction to enter into the execution stage until functional units have finished executing all instructions within the scheduled long instruction.

Figure 1(b-2) shows an execution image for the SVLIW processor to execute SVLIW code. The SVLIW code is constructed by removing all LNOPs from the VLIW code as shown in Figure 1(b-2).

In order to execute the SVLIW code, the SVLIW processor schedules the next long instruction after checking for data dependencies and resource collisions with the scheduled long instructions in advance. When a collision occurs, the processor is stalled as indicated by dash (–) in Figure 1(b-2) until all collisions are resolved. The SVLIW processor uses the same scheduling strategies used for the VLIW processor.

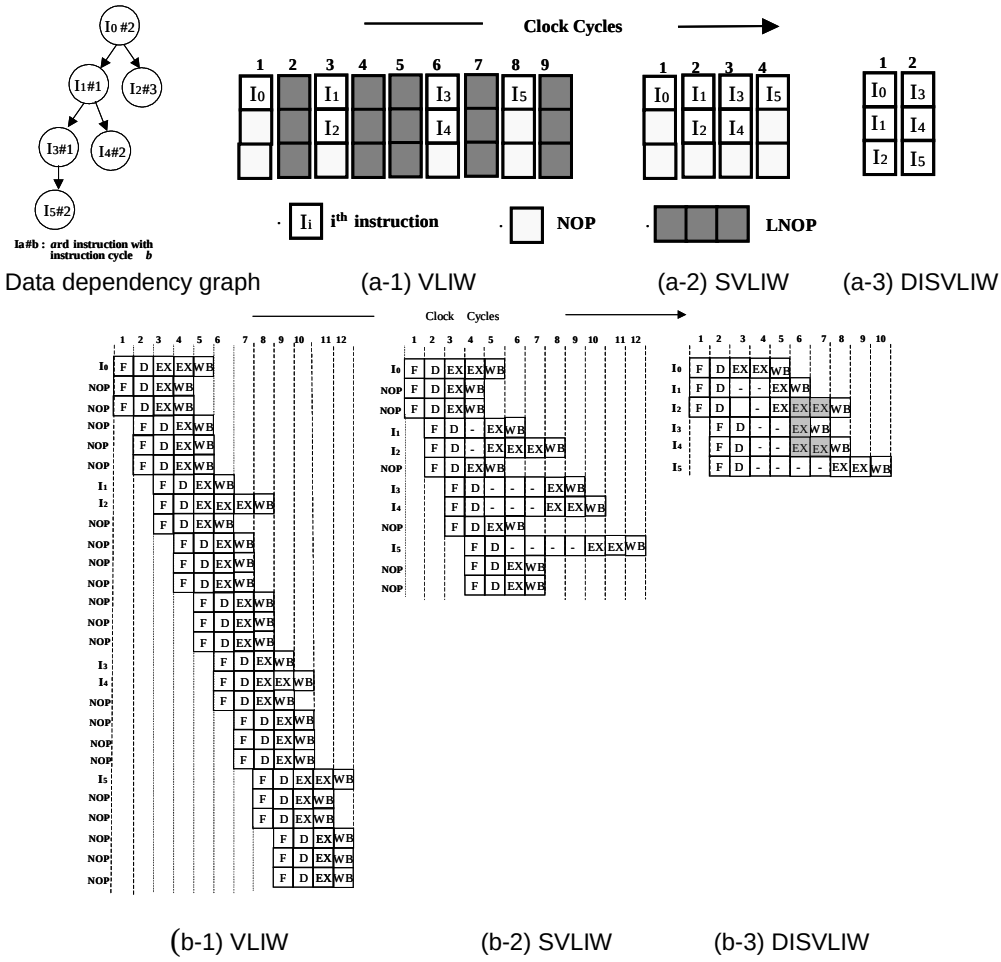


Figure 1. Comparison of issue slots and execution images

Figure 1(b-3) shows an execution image for the DISVLIW processor proposed in this research. Since instructions within a long instruction may depend on each other as shown in Figure 1(a-3), we assume that each instruction contains dependency information in order to achieve synchronization. The DISVLIW processor issues one long instruction per cycle and dynamically executes each instruction using dependency information. As shown in the shaded pipelines in Figure 1(b-3), instructions I_2, I_3, and I_4 are simultaneously executed during the 6th clock cycle although the instructions are fetched on different clock cycle. Instructions I_2 and I_4 are also executed during the 7th clock cycle at the same time.

This example demonstrates the process by which the DISVLIW processor can achieve better performance in comparison to the VLIW or the SVLIW processor. The main insight is that in the DISVLIW processor each instruction within a given long instruction is dynamically processed. Therefore, the DISVLIW processor decreases the waiting time to process a given set of long instructions in comparison to other processors.

3. DISVLIW processor architecture

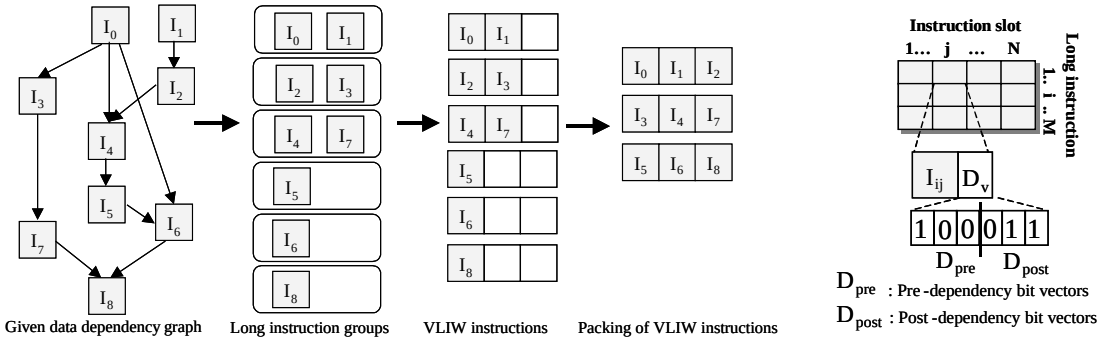
3.1 Long instruction format

To dynamically schedule VLIW instructions, the DISVLIW instruction format is augmented to allow dependent information to be placed in the same VLIW instruction. Dependent information is added to the instruction format to enable synchronization between prior and subsequent instructions.

The problem of optimal DISVLIW code generation can be subdivided into two phases as shown in Figure 2. In the remainder of this paper we will refer to the first phase as *VLIW instruction generation* and to the second phase as *packing*; the result of both phases represents the final DISVLIW code composed of long instructions. Each long instruction has multiple instructions that may depend on each other due to data dependencies or resource collisions.

In the VLIW instruction generation phase, the compiler first generates VLIW code from given data dependency graph where each instruction is assigned to a long instruction as shown in Figure 2. The result is a sequence of long instructions so that one long instruction can be executed per clock cycle without violating data dependences or resource constraints. Empty instruction slots within a long instruction have to be filled with *NOPs* (NOPs are depicted with a white background). In the packing phase, the compiler constructs DISVLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code and by inserting dependency information to each instruction.

To store the dependency relations between instructions, each instruction format consists of an instruction I_{ij} and dependency vector D_V, which has pre-dependency D_{pre} and post-dependency D_{post}. I_{ij} refers to the j^{th} ($j=1,..,N$) instruction within the i^{th} ($i=1,..,M$) long instruction. D_{pre} provides information about functional units executing prior instructions that have dependencies with I_{ij}. D_{post} provides information about functional units that will execute subsequent instructions that depend on I_{ij}. D_{pre} and D_{post} are individually composed of a bit vector that has (N-1) bits, where N equals the number of functional units. To store the information to a bit vector, the compiler allocates one bit for every other functional unit. If I_{ij} depends on a prior instruction I_{lk} ($k<j$ *if* $l=i;k=1,..,n$ *if* $l<i$) being executed by functional unit F_k, the bit designating F_k in the D_{pre} is set to 1. Otherwise, it is set to zero. Although DISVLIW code contains dependency information composed of many bits, the processor can still achieve a reduction in object code size in comparison to the VLIW processor [21]. Figure 2(b) shows the example of DISVLIW code (N=4).



(a) code generation (b) code structure

Figure 2. DISVLIW code generation

3.2 DISVLIW processor implementation

Figure 3 shows the DISVLIW processor architecture. The DISVLIW processor has FU (Functional Unit) and DS (Dynamic Scheduler) pairs, a number of IQs (Instruction Queue) and DCs (Dependency Counter), a register file, an instruction cache, a data cache, and a BTB (Branch Target Buffer). IQs are placed in front of each FU. It seems like instructions within a IQ issue in order, but instructions among IQs slip with respect to each other, dynamic scheduling allows instructions in different IQs(i.e. different FUs) are synchronized by having counters (DC) at each FU. If there are *N* FUs, then each FU has a DC composed of *N-1* counters, 1 counter for every other FU. Each DC saves D_{post} of executed instructions on the associated FU. Using the DC, each DS dynamically decides whether to assign the next instruction to the associated FU, or to stall the FU due to resource collisions or data dependencies. The processor also utilizes the BTB structure for branch prediction [6,9].

Every DS checks for data dependencies and resource collisions among instructions per each cycle using both D_{pre} of the next instruction and counter values in the associated DC. In Figure 4, we assume that the DISVLIW processor has five pairs of FU and DS. In order to schedule instruction, Each DS compares D_{pre} of the next instruction to counter values in the associated DC per each cycle. If any bit in D_{pre} is set to 1, DS checks the counter in the corresponding location in the DC. If the counter is 0, it means that the execution of prior dependent instruction hasn't finished. That is, *d(i)* returns zero. Otherwise, the execution of prior dependent instruction has finished. That is, *d(i)* returns 1. After the DS confirms that the execution of all prior dependent instructions is finished (all of *d(i)* return 1), the DS decrements the counter values in corresponding location in its DC using the set bits in given D_{pre}. It is necessary to clear the D_{post} of the prior instructions from the DC before execution. Simultaneously, each DS individually assigns the next instruction to the associated FU.

As an example of Figure 4, DS_0 checks D_{pre} (1010) of the next instruction and counter values (1030) in its DC_0. Since counters in corresponding position in DC_0 are greater than 0, DS_0 decrements counters in DC_0 using set bits in D_{pre}. As soon as DC_0 turns from (1030) to (0020), then DS_0 assigns the next instruction to FU_0.

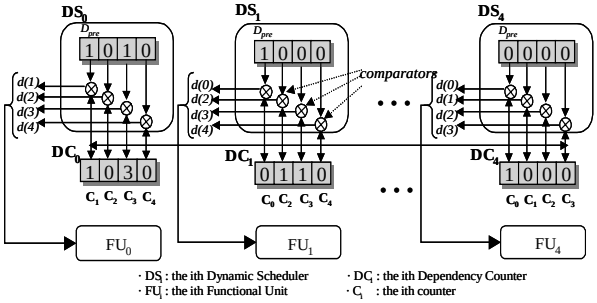


Figure 4. Dynamic scheduler units

3.3 Instruction pipeline algorithm

Each instruction on the DISVLIW processor is executed in four stages as shown in Figure 3. Each stage requires one cycle except the execution stage that requires various execution cycles according to an instruction type. In the Fetch (F) stage, the fetch unit gets one long instruction from the instruction cache each clock cycle and separates it into instructions to store IQs. If IQ is in the full state, the fetch unit cannot fetch the following long instruction, which prevents the IQ from overflowing. In the Decode/Scheduling (D/S) stage, the decode unit analyzes the next instruction at the head of each IQ. Every DS simultaneously checks for data dependencies and resource collisions using both D_{pre} of the next instruction and counter values in the its DC. If there are no data dependencies and resource collisions,

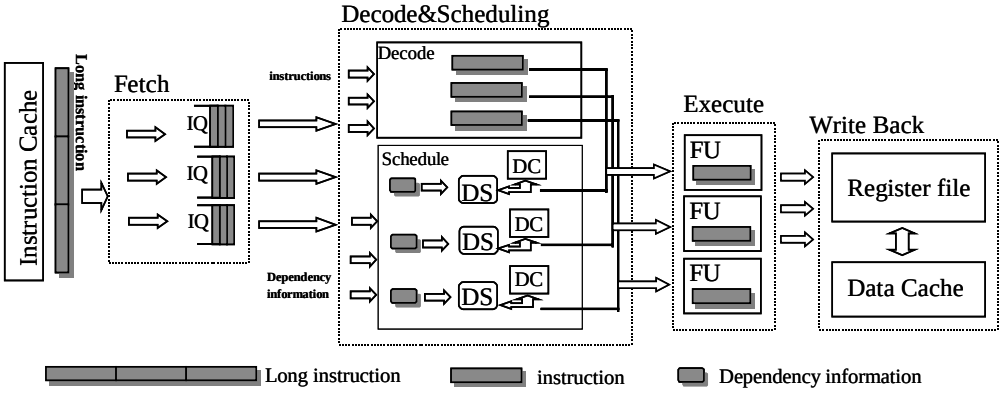


Figure 3. DISVLIW processor architecture

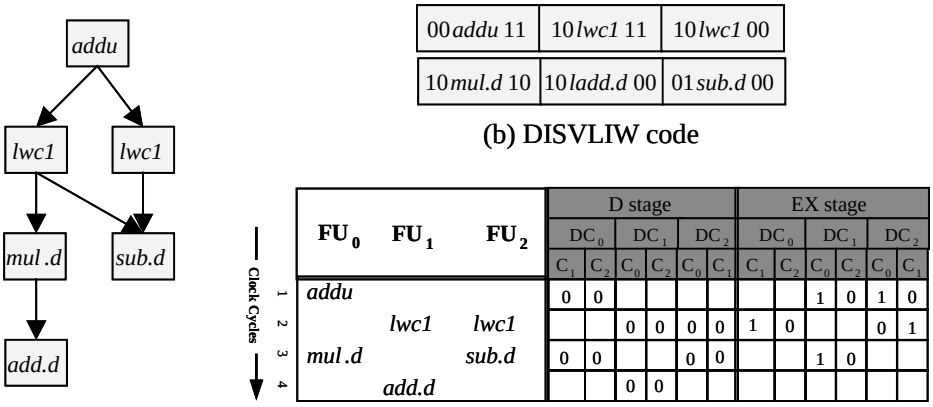


Figure 5. Changes of DC according to instruction execution

each DS decrements counter values in its DC in order to clear the D_{post} of the prior instructions from its DC and assigns the next instruction to the associated FU. In the Execute (EX) stage, every FU executes instruction and announces to other FUs that its execution will be finished during the execution of the final cycle. To accomplish this, the FU increments counters (indicating the FU) in DCs in corresponding location using set bits in the D_{post}. Thus, every FU achieves synchronization since it decrements counter values in its DC at D/S stage and increments it at EX stage. To facilitate this, we designed the EX stage with the ability to control the D/S stage. Finally, in the Write Back (WB) stage, the results of the executed instructions are stored in the register file.

Figure 5 shows execution examples of DISVLIW code generated from Figure 5(a). FU_0 first executes instruction *addu* since the D_{pre} of *addu* is 00, and simultaneously increments the first counters (indicating FU_0) in the DC_1 and DC_2 because the D_{post} of *addu* is 11. Then, FU_1 and FU_2 individually check D_{pre} bits of the next instruction *lwc1* and the counter values in its DC_1 and DC_2. If both of them are greater than 0, FU_1 and FU_2 decrements the first counter value in its DC_1 and DC_2 using set bits in the D_{pre}. It is necessary to clear the D_{post} of the instruction *addu* from each DC before the execution of FU_1 and FU_2. Then, FU_1 and FU_2 simultaneously begin the execution of instructions *lwc1*.

3.4 Loop performance

The DISVLIW processor can significantly reduce the execution cycles of applications containing loops since the processor can simultaneously schedule the instructions fetched from different iterations in the loop.

Figure 6 shows execution images of the VLIW and the DISVLIW processors that execute the i^{th} iteration and the $(i+1)^{th}$ iteration of the loop. We generate VLIW code of Figure 6(a-2) and DISVLIW code of Figure 6(a-3) from the MIPS code of Figure 6(a-1). The MIPS code is to initialize integer array. A long instruction has three instructions each execution cycle of which is one cycle except that those of *mul* are two cycles. Every instruction is executed in the following four pipeline stages: F, D, EX, and WB. Figure 6(b) shows an execution image of the VLIW processor that executes Figure 6(a-2). The VLIW processor requires 15 cycles to execute two iterations of the loop. Figure 6(c) shows an execution image of the DISVLIW processor that executes Figure 6(a-3). Intructions *addu* and *sw* are simultaneously executed for the 6^{th} cycle although the instructions are individually fetched at the 2^{nd} and the 3^{rd} clock cycle. Besides, instructions *blt* and *lw* are simultaneously executed at the 8^{th} cycle although the instructions are fetched from different iterations. The DISVLIW processor requires 14 cycles to execute two iterations of the loop.

From the above observation, we know that the DISVLIW processor is more effective than the VLIW processor in applications containing loops. This is because the DISVLIW processor can simultaneously schedule instructions fetched from different iterations of the loop as long as the instructions don't depend on each other. Due to this feature, the larger the number of loop iterations the DISVLIW processor gets reduced execution cycles in proportion to those, when compared with the VLIW processor. Although is not shown in the Figure 6, the DISVLIW processor can reduce fetch cycles because the DISVLIW processor can simultaneously fetch a number of instructions that may depend on each other in a long instruction. The DISVLIW processor also has relative low cache miss rates due to its reduced object code [21].

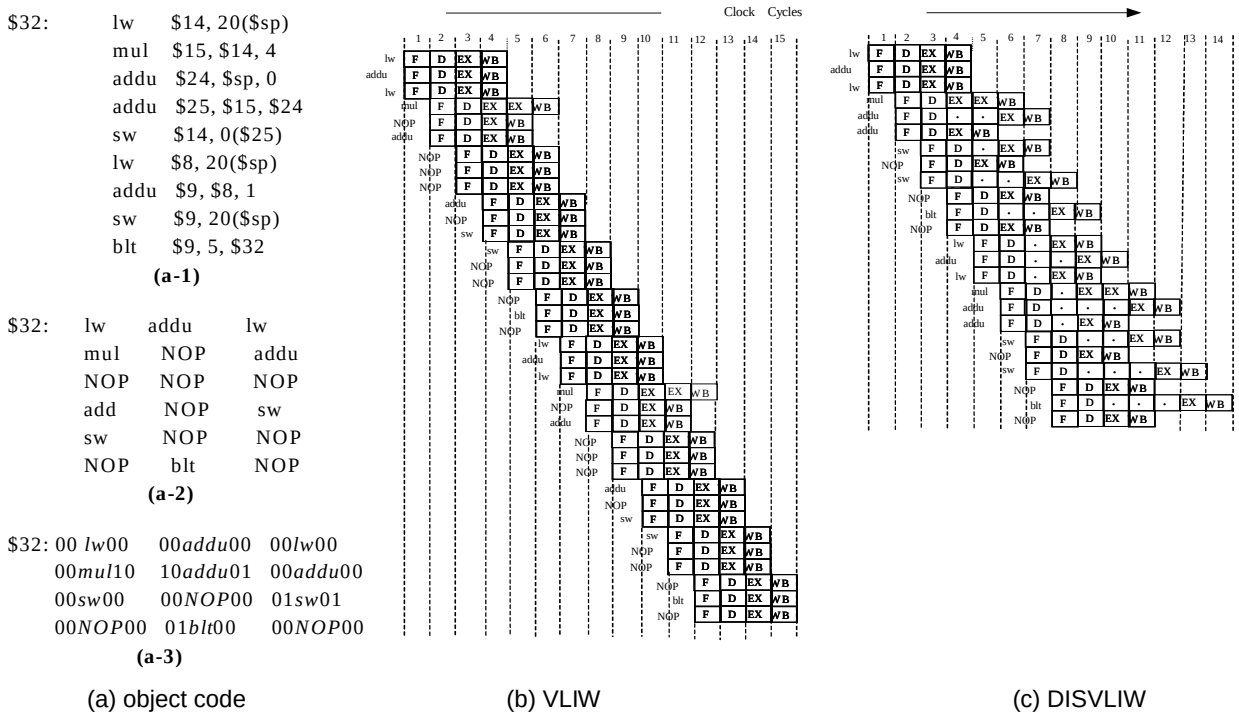


Figure 6. Comparison of loop execution

4. Experiment and analysis

4.1 Simulation system

The performance of the DISVLIW processor was accurately analyzed using a simulator testbed. Using the simulator testbed, we measured the total number of execution cycles for various numerical benchmark applications on the VLIW, the SVLIW, the DISVLIW processor architectures.

The simulator starts with the MIPS assembler, a Mipspro C++ compiler using optimization flag –O and assembly code generation flag –S, generating MIPS R4000 assembly code by compiling a C-language benchmark applications [17]. Next, the macro expander inputs the MIPS R4000 assembly code while simultaneously expanding macros. The Macro expander then passes the assembly code to each parallelizer. Three parallelizers, each of which is associated with a unique processor, are designed with the ability to exploit ILP across basic blocks using compile techniques such as register renaming, branch prediction, invariant code motion from loops, common subexpression elimination, function inlining, and loop unrolling [6,9,10,11]. Generally, the VLIW's effectiveness depends on how good the compiler is: the VLIW processor using a compiler with higher ILP will produce better performance, and will get higher cache hit rates because of the reduced object code size. However, the DISVLIW processor accomplishes this same goal since it constructs object code using the VLIW code. In the diagram, $VLIW_C$, $VLIW_S$, and $VLIW_{DIS}$ correspond to VLIW, SVLIW, and DISVLIW code, respectively. The parallelizers then use the MIPS code to generate parallelized code for its processor simulator and then translate this parallelized code into object code.

For these experiments, processor speedups are calculated by dividing the total number of execution cycles of the VLIW processor by the total number of cycles of the DISVLIW or the SVLIW processor. In the Table 1, the fixed parameters and the variable parameters are also shown. Except when stated otherwise, the default values were used in the simulations.

Table 1. Input parameters

Fixed Parameters	
Processor pipeline	Four-stage(F,D,EX, WB)
Decoded instruction size	4 bytes
integer instruction latency	1 cycle
Floating point instruction latency	1~32 cycle(depend on instruction)
Data cache size	Perfect(no miss penalty)
cache mapping method	direct mapped
cache replacement policy	LRU(Least Recently Used)
Variable Parameters	
Parameter	**Default Value**
A number of integer unit	2
A number of floating-point unit	2
next long instruction miss penalty	4
Instruction cache size	16k bytes

Table 2 provides the benchmark applications and the proportion of I/F (Integer instructions and Floating-point instructions) of each benchmark application. These applications all use double precision.

Table 2. Benchmark applications

Benchmarks	Description	I/F(%)
LIVERMORE	Do loop for various kernel operations	65.3/34.7
MM	Matrix Multiply using floating point instructions	68.4/31.6
CLINPACK	Set of linear algebra subroutine	75.7/24.3
WHETSTONE	Loop instructions for arithmetic computation	65.6/34.4
FFT	Matrix Fourier Transformation	43.3/56.7

Table 3 tabulates the ratios of object code size of the VLIW to both the SVLIW and DISVLIW processors for each benchmark. In this experiment, we chose numerical benchmarks that have a high proportion of floating-point instructions. This choice was appropriate because the DISVLIW processor is more effective given dynamic instruction scheduling and reduced object code size. Even though $VLIW_{DIS}$ contains many bits of dependency information, Table 3 indicates that $VLIW_{DIS}$ averages 45% smaller than $VLIW_c$ and is almost the same size as $VLIW_s$.

Table 3. Comparison of object code size

Benchmarks	$VLIW_C$	$VLIW_S$	$VLIW_{DIS}$
LIVERMORE	1	0.723	0.725
MM	1	0.568	0.591
CLINPACK	1	0.673	0.673
WHETSTONE	1	0.438	0.385
FFT	1	0.385	0.400
AVERAGE	1	0.557	0.554

4.2 Experimental results

Figure 7 shows the speedup of the DISVLIW processor over the VLIW (or the SVLIW) processor using different scheduling strategies. In order to evaluate scheduling performance only, we ignore cache effects such as cache miss rates. We assume that an instruction cache size is perfect (no miss penalty). In this experiment, we reduced the number of loop iterations in each benchmark application to reduce simulation duration.

Figure 7 illustrates that even though we assume a cache with a zero miss rate, the DISVLIW's performance is still 9%-15% higher than that of the VLIW processor regardless of benchmark application. We have the DISVLIW's scheduling strategies to thank for this speedup. This scheduling decreases the waiting time to process a set of long instructions when compared to the VLIW and SVLIW processors. By contrast, the VLIW and the SVLIW processor can't execute pending long instructions until the execution of all instructions in the previous long instruction finishes. In Figure 7, the SVLIW processor shows same performance when compared to the VLIW processor.

Figure 8 illustrates the impact of cache size on speedups of the DISVLIW processor with respect to both the SVLIW and VLIW processors. We varied the instruction cache size from 4k bytes to 32k bytes to compare performance according to changes in cache size. The speedups of the DISVLIW and the SVLIW processors were measured relative to the VLIW processor regardless of cache size. In this experiment, we also reduced the number of loop iterations in each benchmark to reduce simulation duration.

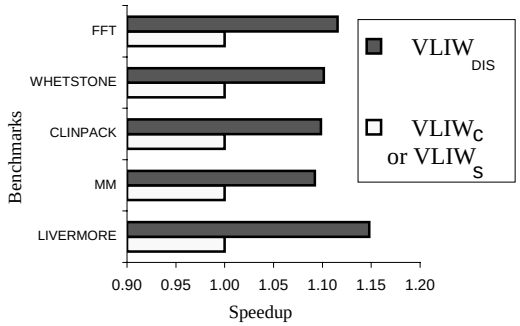


Figure 7. Comparison of speedups for different scheduling strategies

These results indicates that the DISVLIW processor is faster than the SVLIW processor regardless of both benchmark applications and cache size. This is due to the DISVLIW's unique instruction scheduling strategies. Another factor is the DISVLIW's reduced object code size, which decreases average fetch cycles and also reduces cache misses, as shown in Table 3. Figure 8 indicates that larger cache sizes result in smaller speedup differences between the VLIW and DISVLIW processors. At smaller cache sizes, the VLIW's performance is slower due to higher cache miss rates. Unlike the VLIW, the DISVLIW's performance is not as sensitive to cache size due to its smaller object code. But as cache size increases, performance difference decreases and the VLIW's performance approaches that of the DISVLIW. Yet, even assuming perfect cache, the DISVLIW is still faster than the VLIW's.

Overall, the performance of DISVLIW processor is faster than the VLIW and the SVLIW processors over a wide range of cache size and across various numerical benchmark applications. We attribute these performance gains to the balanced benefits of compile-time and run-time parallelization, dynamic instruction scheduling, and size reduction of object code as previously described.

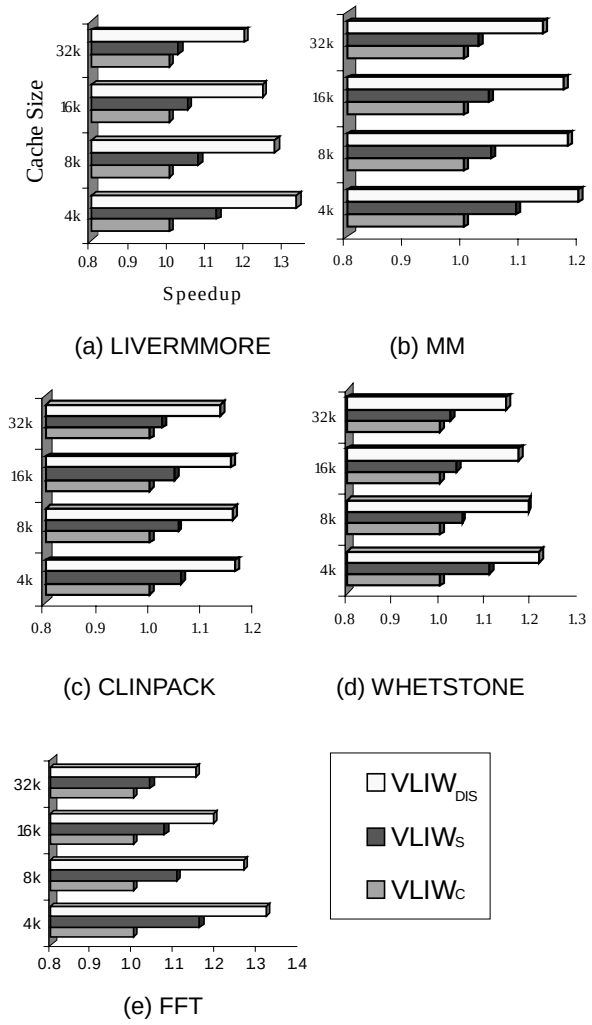


Figure 8. Comparison of the speedups according to changes in cache sizes

5. Conclusion

This paper describes a new ILP processor architecture referred to as Dynamically Instruction Scheduled VLIW (DISVLIW). The DISVLIW processor is a hybrid architecture that has inherited features as ILP exploitation at compile-time of the VLIW processor and dynamic scheduling at run-time of the superscalar processor. The experimental evaluations presented in this paper have shown that the DISVLIW processor achieves a high speedup over the VLIW and the SVLIW processors for a wide range of cache sizes and across various numerical benchmark applications. These performance gains of the DISVLIW processor result from dynamic instruction scheduling and size reduction of object code.

The DISVLIW processor architecture opens several new avenues of research. Optimization of dependency information within object code, DISVLIW compilers, and scalability of functional units in the system are just a few examples that will be investigated in future work. Particularly, our research will focus on optimization and management of the dependency information required in order to achieve synchronization.

6. References

[1] Ken Sakamura, '21st-century microprocessors,' IEEE Micro, pp.10~11, July/Aug 2000.

[2] Michael J. Flynn, Computer Architecture, Jones and Bartlett Publishers, 1995

[3] P. P. Chang, D. M. Lavery, S. A. Mahlhe, W. Y. Chen, and Wen-Mei. W. Hwu, “The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors,” IEEE Transactions on Computers, Vol. 44, No. 3, pp. 353~370, March 1995.

[4] Shyh-Kwei Chen, W. Kent Fuchs, and Wen-Mei W. Hwu, "An analytical approach to scheduling code for superscalar and VLIW architectures," Proc. International Conference on Parallel Processing., pp. I258-I292, 1994.

[5] J. A. Fisher, The VLIW machine: A multiprocessor for compiling scientific code, IEEE Transactions on Computers, pp. 45~53, July 1984.

[6] Barry Fagin, "Partial Resolution in Branch Target Buffers'" IEEE Computers, Vol. 46, No. 10, October 1997

[7] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers., Vol. C-30, No. 7, pp. 478~490, July 1981.

[8] Roger Espasa and Mateo Valero, "Exploiting instruction- and data-level parallelism," IEEE Micro, Vol. 17, No. 5, Sept 1997.

[9] S. A. Mahlke, R. E. Hank, J. E. M. McCormick, D. I. August, and W. W. Hwu, “A Comparison of Full and Partial Predicated Execution Support for ILP Processors,” Proceedings of the 22th international Symposium on Computer Architectures, pp. 138~150, 1995.

[10] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture," Proceedings of 28th International Symposium on Microarchitecture, March 1995.

[11] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," Transactions on Parallel and Distributed Systems, Vol. 5, No. 6, pp. 658~664, June 1994.

[12] T. M. Conte and S. W. Sathaye, “Dynamic rescheduling; a technique for object code compatibility in VLIW architecture,” proceedings of the 28th Annual International Symposium on Micro architecture, pp. 208~218, March 1995.

[13] Kevin W. Rudd and Michael J. Flynn, "Instruction-level parallel processors-dynamic and static scheduling tradeoffs," Proc. The Second AIZU International Symposium on Parallel Algorithms/ Architecture Synthesis., pp. 74~80, March 1997.

[14] Shusuke Okamoto and Masahiro Sowa, "Hybrid processor based on VLIW and PN-Superscalar," Proc. DPTA'96 International Conference., pp. 623~632, 1996.

[15] Sunghyun Jee and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor," Journal IEEE Korea Council, Vol. 1, No. 1, December 1997.

[16] Susan J. Eggers, Joel S. Emer, Henry M. Levy, and Jack L. Lo, "Simultaneous multithreading," IEEE Micro, Vol. 17, No. 5, Sep 1997.

[17] MIPS R4000 Microprocessor User's Manual, MIPS Computer Systems, Inc., 1991.

[18] B. R. Rau, “Dynamically scheduled VLIW processors,” Proceedings the 26th Annual International Symposium on Microarchitecture, pp. 138~148, Mach 1997.

[19] T. Hara and H. Ando, “Performance comparison of ILP machines with cycle time evaluation,” Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 213~224, Mach 1996.

[20] A. F. de Souza and P. Rounce, “Dynamically Scheduling VLIW instructions,” Journal of Parallel and Distributed Computing, pp. 1480~1511, 2000.

[21] Sunghyun Jee and Sukil Kim,” A Design of A Processor Architecture for Codes With Explicit data Dependencies,” Proc. tenth SIAM Conference on Parallel Processing for Scientific Computing 2001, March 2001.

[22] Patterson D. A., and J. L Hennessy, *Computer Architecture A Quantitive Approach 2nd edition*, Morgan Kaufmann, pp. 240~261, 1996.

[23] Intel, http://www.intel.com/ia64, *IA-64 Architecture Software Developer’s Manual, Volume 1:IA-64 Application Architecture, Revision 1.1*, July 2000.

[24] Intel, http://www.intel.com/ia64, *Itanium Processor Michroarchitecture Reference for Software Optimization*, Aug. 2000.

Accuracy of Profile Maintenance in Optimizing Compilers

Youfeng Wu
Programming Systems Research Lab
Intel® Corporation
2200 Mission College Blvd
Santa Clara, CA 95052
youfeng.wu@intel.com

Abstract

Modern processors rely heavily on optimizing compilers to deliver their performance potentials. The compilers on the other hand rely greatly on profile information to focus the optimization efforts and to better match the generated code with the target machines. Maintaining the profile in optimizing compiler is important as many optimizations can benefit from the profile information and they often are performed one after the other. Maintaining a profile is however tedious and error-prone. Erroneous profile is not easy to detect as it affects only the performance, not the correctness, of a program. Maintaining a profile also inherently loses accuracy, as the profile update operations often have to use probabilistic approximation. In this paper, we measure the accuracy of maintaining CFG profiles in a high-performance optimizing compiler. Our data indicates that the compiler maintains the profile more accurately within individual functions than globally across functions, and function inlining may be responsible for the loss of profile accuracy globally. We also identify a list of important research issues related to profile maintenance.

1. Introduction

Modern processors rely heavily on optimizing compilers to deliver their performance potentials. The compilers on the other hand rely greatly on profile information to focus the optimization efforts and to better match the generated code with the target machines [4] [5] [7] [12] [20]. A profile about a program is a relative ranking of the components in the program, collected from previous runs of the same program. The most widely used profile is the control flow profile [1] [2] (referred to as the CFG profile in this paper). A CFG profile provides the compilers with the execution frequency and the branch probabilities for each basic block in the control flow graph. This profile is easy to obtain and provides significant performance boost to optimizing compilers. A recent study [14] shows that CFG-profile guided code layout can improve OLTP performance by 33% on Alpha systems. Our experience indicates that using CFG profile information can lift the performance of the CPU2000 integer suite on Itanium™ systems [9] significantly.

Maintaining a profile during compilation is important. Many optimizations can benefit from the profile information and they often are performed one after the other. During an early optimization, the control flow may change and a reasonably well-maintained profile must be presented to the late optimizations. We could avoid the maintenance issue by recollecting the profile after each optimization. However, profiling a program multiple times imposes huge burden on the productivity and is often impractical.

Maintaining a profile during compilation is however tedious and error-prone. For any transformation that may modify the control flow, additional code must be added to update the profile information. Any omission in the profile maintenance may lead to erroneous profile information later. Erroneous profile information is not easy to detect as it affects only the performance, not the correctness, of a program. For example, a programmer, who may be under time pressure to fix a correctness bug, could easily forget adding the profile maintenance operations related to the bug fixes. One of the benefits of this study is to automatically detect the profile deterioration, and provides hints on where the accuracy loss occurs (e.g. which functions lose profile accuracy most seriously).

Maintaining a profile during compilation also inherently loses accuracy. When the program control flow graph changes, there is often not enough information to update profile information precisely. The most commonly used method is to use probabilistic approximation. Namely, the branch probabilities of the new blocks are assumed to be the same or closely related to the branch probabilities of the corresponding old blocks. For example, Figure 1 shows the profile maintenance for a code replication optimization. The block frequency is shown to the right of each block, and the branch probability is marked on the branching edge. When blocks c, d and e in Figure 1 (a) are replicated, the

new blocks c', d', and e' need to be assigned block frequencies and branch probabilities. Under the assumption that the block c' has the same branch probability as the old block c, the block frequencies can be assigned as shown in Figure 1 (b). The updated profile could be different from the actual execution, however, as the situation in Figure 2 could occur when the duplicated code is executed. In this case, the branch probabilities of block c' are .6 and .4, different from those of the original block c (.3 and .7).

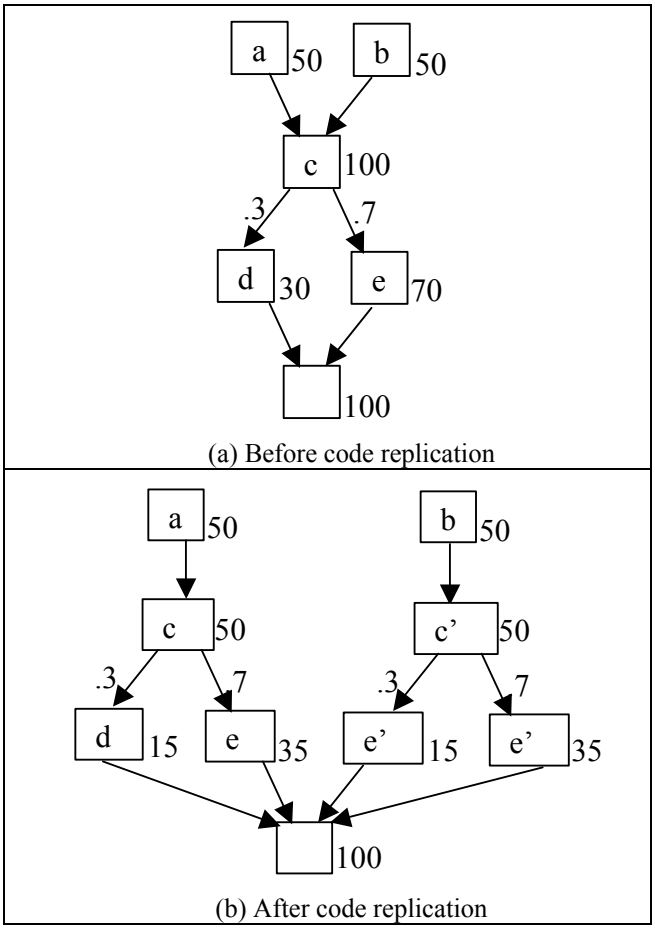


Figure 1. Maintaining the Profile during Code Replication

To our knowledge, there is no previous study of the profile maintenance issue in literature. As the modern processors increasingly rely on profile information to deliver performance, understanding the accuracy of profile maintenance becomes important. In this paper, we measure the accuracy of maintaining a CFG profile in a research compiler for the Itanium Processor Family (IPF). The compiler is based on a production high-performance optimizing compiler with additional components to make compiler and architectural exploration easier. Our results indicate that the compiler maintains profiles more accurately within individual functions than globally across functions. We also identify a list of important research issues related to profile maintenance.

The rest of the paper is organized as follows. Section 2 discusses the approaches for profile maintenance. Section 3 outlines a few optimizations and how they maintain CFG profiles. Section 4 describes the methodology for measuring the profile accuracy. Section 5 provides the experimental results. Section 6 concludes the paper and discusses the future directions.

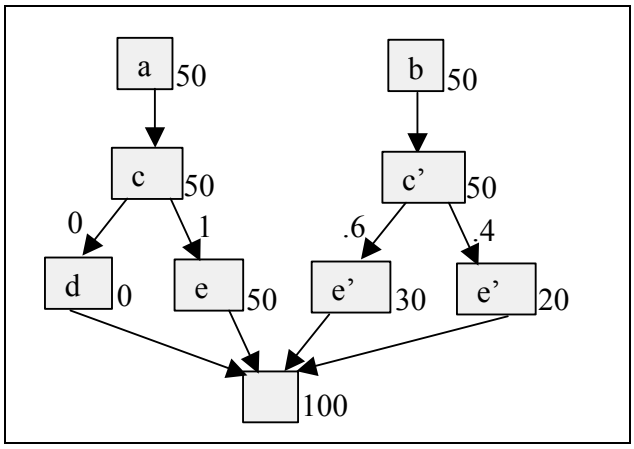


Figure 2. A Possible Execution Situation Different from the Maintained Profile

2. Approaches for profile maintenance

When an optimization changes a program control flow graph, the program's profile needs to be maintained for the subsequent optimizations. The maintained profile must satisfy the *rule of flow conservation*. Namely, for every basic block in the control flow graph, the total frequency on its incoming edges is the same as the total frequency on its outgoing edges. The frequency of the block is the same as its total incoming frequency or its total outgoing frequency. The frequency for a control flow edge is the product of the frequency of its source block multiplied by the branching probability along the edge.

Occasionally, using the rule of flow conservation alone can uniquely update the block frequencies for a simple optimization, as shown in Figure 3. In general, however, there are many ways to maintain a profile by following the rule of flow conservation. For examples, the profile maintenances in Figure 1 (b) and Figure 2 both follow this rule, although the updated profile in Figure 1

(b) is preferred, as it is more likely to happen in actual execution.

In order to obtain the more likely maintained profile, the profile maintenance should try to derive the branch probabilities for the updated blocks from those for the corresponding blocks in the original control flow graph. We call this rule the *probabilistic approximation*. The profile maintenance in Figure 1 (b) follows the probabilistic approximation by assuming the block c' having the same branch probability as block c, while that in Figure 2 does not. In cases when the branch probabilities for the new blocks cannot be the same as the original blocks, there are often simple heuristics to determine the new branch probabilities from the original branch probabilities. As long as a branch probability assignment can be determined for the updated control flow graph, a unique maintained profile can be obtained following the rule of flow conservation. Maintaining a profile using the two rules tends to change the profile slowly and provides a reasonably good profile for late optimizations.

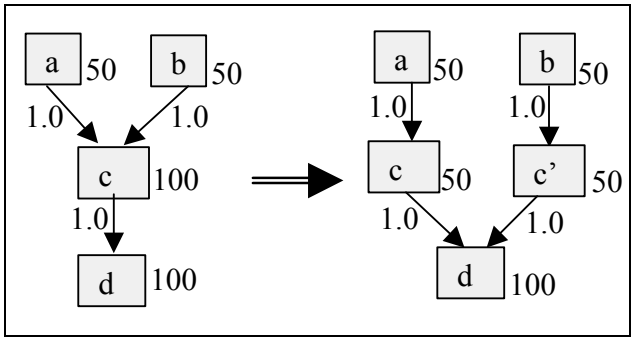


Figure 3. Frequencies for c and c' can be uniquely determined by rule of flow conservation

The above two rules suggest the following two-step profile maintain procedure.

1. Use probabilistic approximation to assign branch probabilities to the updated basic blocks.
2. Propagate block frequencies following the rule of flow conservation.

The first step is usually straightforward. The branch probabilities for a new basic block are either the same as those for the original basic block, or slightly modified from those of the original basic block. Optimizations that may require modified branch probabilities from the original basic block including the following:

Branch reversal optimization: need to change the branch probability from p to 1-p

Counted loop unrolling: need to change loop exit probability from p to p*unrolling_factor.

Once the branch probabilities are assigned, the second step can be performed using the frequency propagation procedure in [19] (i.e. the **Algorithm 2** in [19]**)**. The algorithm takes the frequency of function entry block (or a region entry block) and branch probabilities for the blocks in the function (or a region) as input, and determines the frequencies for every basic block in the function (or region). The resulting block frequencies conform to the rule of flow conservation and also are consistent with the branch probability assignment.

The second step can be performed either globally or incrementally, depending on the optimizations. If an optimization does not use the profile information in the modified control flow graph, a global application of the frequency propagation algorithm can be applied to the entire function. If an optimization repeatedly transforms a region of code and requires the profile information for the modified region of code be up to date, the optimization needs to propagate block frequencies incrementally after optimizing each region of code. Our compiler uses the incremental approach to maintain profiles for most optimizations.

3. Optimizations and profile maintenance

In this section, we outline a partial list of optimizations in our compiler that uses and maintains the CFG profile. The following optimizations change control flow graph and need to maintain the CFG profile after using it.

Function inlining: replace a function call with the function body to enlarge optimization scope and eliminate call overhead. This optimization is often applied to call that is frequently invoked [6].

Function cloning: specialize a function with the constant parameters from a specific call site. This optimization is often applied to a function call with constant parameters and high execution frequency [17].

Loop unrolling: replicate a loop body multiple times to enlarge loop body and enable other optimizations, such as software pipelining and instruction scheduling. This optimization is often applied to loops with high trip count. The trip count can be obtained from the frequencies of the loop pre-head block and the loop entry block [15][11].

Loop peeling: peel a few iterations from a loop so the peeled code can be optimized with the surrounding code. This optimization is often applied to an inner loop with a low trip count [8].

Tail duplication: duplicate a control flow subgraph to remove infrequent side entries to a code region [8].

Switch optimization: convert a switch statement implemented with an indirect branch to a series of direct branches. This optimization is often applied when the

indirect branch has a few targets that are taken much more frequently than the other targets.

Branch optimization: combine or remove constant conditions to short circuit branch chains [16].

Code placement: arrange code so that instruction flow is only infrequently interrupted. It places blocks that are frequently executed following each other together in the generated code [13]. This optimization may reverse the branch directions for a basic block and may change the branch probabilities.

The following optimizations benefit from the CFG profile and usually do not change control flow so need not to maintain the CFG profile.

Loop invariant code hoisting: hoist operations that are invariant inside the loop body to the loop pre-head block. This optimization is often applied when the invariant operations are executed more frequently than the loop pre-head block.

Instruction scheduling: arrange instructions to match machine resource so to minimize the execution time. The block frequency information allows the scheduler to focus on the most important traces, possibly sacrificing less frequent code. The frequency information is also crucial for control and data speculation [9][3].

Register allocation: allocate physical registers to program variables. Block frequency information allows the register allocator to reduce register spill overhead [10].

In the rest of the section, we discuss in slightly more detail the function inlining and the loop unrolling optimizations and show how they use and maintain CFG profile.

3.1. Function inlining

Function inlining is a sequence of decisions to inline a selected set of calls to maximize overall performance of the program. One of the decision criteria to inline a call is that the call is invoked in a block with high frequency. After a function *g* is inlined into a function f, the function g will no longer be called from the call site. Thus, the frequencies of the blocks in function *g* must be reduced to reflect the fact that the function will be called less often after the inlining. In addition, the inlined copy of the function g inside function f must be assigned block frequencies and branch probabilities. In essence, function inlining is a special case of code replication: the function body is replicated into the caller function. The profile can be maintained as follows.

Assume the entry block frequency of function *g* is F1, and the frequency of the basic block of the call site is F2. For each block b1 in *g* and the corresponding block b1' in the inlined copy of *g*, the frequency maintenance is performed as follows:

old_freq = freq(b1)
freq(b1') = old_freq * F2/F1
freq(b1) = old_freq –freq(b1')

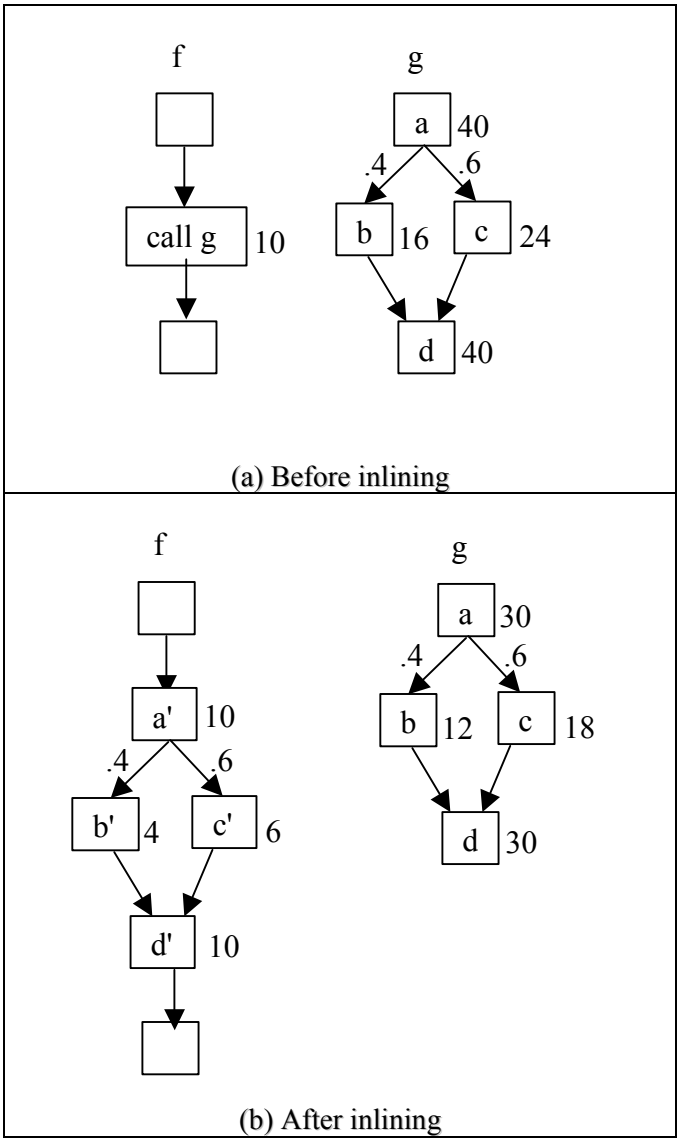


Figure 4. Maintaining a profile for function inlining

Look at the example shown in Figure 4. F1 = 40 and F2 = 10. The frequencies for the blocks of g inlined into f are computed as follows:

freq(a') = 40 * 10/40 = 10;
freq(b') = 16 * 10/40 = 4;
freq(c') = 24 * 10/40 = 6;
freq(d') = 40 * 10/40 = 10.

The frequencies for the blocks inside function g are updated as follows:

freq(a) = 40 - freq(a') = 30;
freq(b) = 16 - freq(b') = 12;
freq(c) = 24 - freq(c') = 18;
freq(d) = 40 - freq(d') = 30.

The operations above assume that each block in the inlined copy has the same branch probabilities as the corresponding block in the function before inlining. This clearly can result in loss of profile accuracy. For example, different calls may exercise different portions of a function during the actual execution. Even if the same portion of a function is executed for different calls, the execution may have significantly different branch probabilities.

3.2. Loop unrolling

Loop unrolling replaces the loop body by multiple copies of the original loop body. The number of copies is usually referred to as the *unrolling factor*. The unrolling factor can be determined using the block frequencies information. For example, one of our methods determines the unrolling factor from the trip count of the loop. The trip count of the loop is computed as follows.

$$\text{Trip\_count} = \frac{\text{freq(loop entry block)}}{\text{freq(loop pre\_head block)}}$$

When a counted loop is unrolled, the unrolled loop body will not have an early exit. The block frequency can be maintained by dividing the frequency of each block in the original loop body by the unrolling factor, as shown in Figure 5 (a) and (b), where a1 and a2 are copies of block a. Notice that, in this case, the new block a1 has a branch probability of 1.0 to a2, which is different from the branch probability of the original block a branching to itself (0.9). Also, the loop exit probability is increased by the unrolling factor as the number of times the back edge is taken is reduced by the unrolling factor.

For a while loop (or a do-while loop), the unrolled loop body contains early exits. The blocks in the early copies of the original loop body in the unrolled loop body should have higher frequency than the corresponding blocks in the later copies. Following the rule of probabilistic approximation, we may assumes that the exits in the unrolled loop body all have the same taken probability as the loop exit probability in the original loop. With this assumption, the block frequencies for the example shown in Figure 6 (b) can be calculated using the frequency propagation procedure [19] as follows:

freq(a1) = 10/(1-0.9*0.9) = 53
freq(a2) = freq(a1) * 0.9 = 47

The resulting profile updated by following the rule the probabilistic approximation may work better than simply dividing the block frequency by the unrolling factor for while loops. However, some while loops are just disguised version of counted loops. In this case, the "probabilistic approximation" assumption may actually work less well than dividing the block frequency by the unrolling factor. Our compiler currently dividing the block frequencies by the unrolling factor for both counted and while loops.

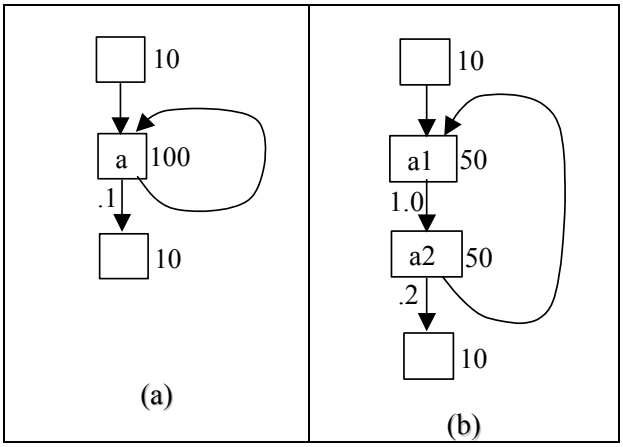


Figure 5. Maintaining the profile for a counted loop with unrolling factor of 2

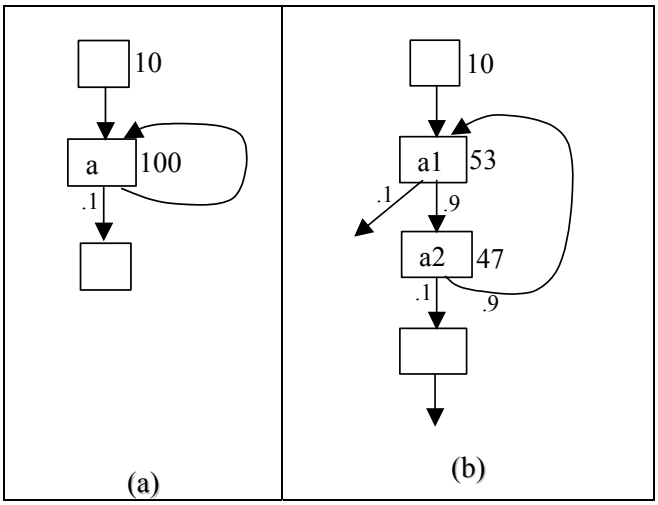


Figure 6. Maintaining a profile for a do-while loop with an unrolling factor of 2

4. Experimental methodology

The compiler collects an early CFG profile in the beginning of the compilation process and maintains it

throughout the optimizations. We will call the profile obtained after profile maintenance the "maintained profile". To measure the accuracy of the profile maintenance, we collect an additional profile after all the optimizations that may change control flow are performed. We will call the new profile the "accurate profile". We compare the maintained profile and the accurate profile after all the optimizations that may change control flow are performed. If the maintained profile is maintained accurately during optimizations, it should be compared the same as the accurate profile. This experiment uses a three-pass compilation process as shown in Figure 7 to compare the profiles.

PASS 1	PASS 2	PASS 3
• Compile and run the program to collect an early profile	• Feedback the early profile • Perform optimization & maintain the early profile • Run the optimized program to collect the accurate profile	• Feedback the early profile • Perform optimization & maintain the early profile • Feedback the accurate profile • Compare maintained profile and accurate profile

Figure 7. Process for comparing maintained profile with accurate profile

We use Wall's weighted and unweighted matching method [18] to compare the block frequencies in the maintained profile and the accurate profile. Assume that a program (for global matching) or a function in a program (for local matching) contains N blocks. The maintained and accurate profiles may have different frequencies for the blocks. We denote the frequency of a block blk in the accurate profile with afreq(blk) and the frequency in the maintained profile with mfreq(blk). We first generate two sorted lists of the N blocks with the highest frequency first, one called accurate_list, sorted by their frequencies in the accurate profile, and another called maintained_list, sorted by their frequencies in the maintained profile. We compare the top m blocks in the two lists and identify the set of blocks in the top m blocks in the maintained_list that also occur in the top m blocks in the accurate_list. Let the set of matched blocks be *matched(m)*. The unweighted matching score is computed by the following formula:

$$\frac{|matched(m)|}{m}.$$

The weighted matching score is computed by the following formula:

$$\frac{\sum_{b \in \text{matched(m)}} afreq(b)}{\sum_{i=1}^{m} afreq(b_i)}$$

A perfect match will have a score of 100%. A random sorting of the maintained_list will have a likely score of m/N. The closer to 100%, the more accurate is the profile maintenance.

In our experiment, we calculate matching scores for the top 10% to top 50% of the blocks (i.e. for m equals to 10%*N to 50%*N). In general, if less than 80% of the top 10%*N of the blocks in the maintained list is in the top 10%*N of the blocks in the accurate list, we may say that the profile is NOT maintained accurately. As a reference, the block frequency information obtained from static heuristics has a matching score approaching 80% for the top 10% blocks [19]. In other words, if the matching score for the maintained profile is less than 80% for the top 10% of blocks, we may be wasting our effort to maintain the profile. We could simply use static heuristics to estimate the block frequencies after each optimization. On the other hand, if 90% or more of the top 10% of the blocks in the maintained list is also in the top 10% of the blocks in the accurate list, we may say that the profile is maintained accurately.

We present the matching scores collected globally or locally. The global matching scores are collected by comparing the maintained_list and the accurate list for the entire program. The local matching scores are the average of the scores collected by comparing the two lists within individual functions. Maintaining global profiles accurately is important for many inter-procedural optimizations, such as procedure placement. Maintaining local profiles accurately is important for many function local optimizations, such as loop optimizations and code motion transformations.

5. Experiment results

Our experiment is performed in a research compiler for the Itanium Processor Family (IPF). The compiler is based on a production compiler with additional components to make compiler and architectural exploration easier. The production compiler team has spent significant effort to enhance the algorithms for profile maintenance and resolve related programming bugs. We believe that the compiler has done nearly all possible to maintain a profile accurately and it should represent the state-of-art technology in profile maintenance. Our focus is to measure the inherent loss of profile accuracy during compiler optimizations.

In this experiment, the compiler uses the base option set for maximal performance. The base option set

includes all of the optimizations we mentioned in Section 3. We use the CPU2000 integer benchmarks shown in Figure 8 running with the train input set to collect and compare the profiles. We only compare blocks with frequencies greater than a threshold, such as 500.

PROGRAMS	DESCRIPTION
164.gzip	Compression/Decompression
175.vpr	FPGA circuit placement and routing
176.gcc	C programming language compiler
181.mcf	Combinatorial Optimization
186.crafty	Game Playing: Chess
197.parser	Word Processing
252.eon	Computer visualization
253.perlbmk	PERL programming language
254.gap	Group theory, interpreter
255.vortex	Object-oriented database
256.bzip2	Compression
300.twolf	Place and route simulator

Figure 8. CPU2000 integer benchmarks

5.1. Global matching scores

Figure 9 shows the weighted and unweighted global matching scores. We first look at the unweighted scores. On the average, about 85% of the top 10% of blocks in the maintained list remain in the top 10% of blocks in the accurate list. This average score is a little low. A few benchmarks, such as 175.vpr and 181.mcf, show even worse scores. For example, the compiler maintains only 70% and 57% of the top 10% blocks for the two benchmarks, respectively. The low global score may affect global optimizations, such as, procedural placement.

The weighted and the unweighted global scores are noticeably different for the 255.vortex benchmark. Its unweighted matching score for the top 10% blocks is 93%, while its weighted matching score for the top 10% blocks is only 81%. Similarly for the 181.mcf benchmark: its unweighted matching score for the top 50% blocks is 88%, while its weighted matching score for the top 50% blocks is only 74%. Overall, the weighted global matching scores are slightly lower than their unweighted counterparts. This indicates that the global loss of profile accuracy happens more often to the highly frequent blocks than to the infrequent blocks, probably because that more optimizations that can change control flow are applied to frequent blocks.

	unweighted					weighted				
Global	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
164.gzip	0.90	0.95	0.94	0.95	0.97	0.92	0.93	0.93	0.93	0.98
175.vpr	0.70	0.86	0.91	0.94	0.96	0.68	0.81	0.88	0.88	0.89
176.gcc	0.92	0.94	0.94	0.96	0.97	0.92	0.93	0.94	0.95	0.95
181.mcf	0.57	0.73	0.82	0.85	0.88	0.67	0.71	0.73	0.73	0.74
186.crafty	0.87	0.91	0.91	0.92	0.92	0.88	0.91	0.93	0.93	0.94
197.parser	0.86	0.88	0.94	0.96	0.96	0.84	0.85	0.89	0.89	0.96
252.Eon	0.85	0.89	0.94	0.96	0.95	0.83	0.90	0.94	0.97	0.97
253.perlbmk	0.81	0.87	0.92	0.96	0.98	0.87	0.91	0.95	0.98	0.99
254.gap	0.93	0.98	0.98	0.99	0.99	0.91	0.98	0.99	1.00	1.00
255.vortex	0.93	0.99	1.00	1.00	1.00	0.81	0.98	1.00	1.00	1.00
256.bzip2	0.94	0.88	0.90	0.89	0.93	0.97	0.94	0.93	0.93	0.94
300.twolf	0.96	0.98	0.99	0.99	0.99	0.98	1.00	1.00	1.00	1.00
geomean	0.85	0.90	0.93	0.95	0.96	0.85	0.90	0.92	0.93	0.94

Figure 9. Global Matching Scores

	unweighted					weighted				
Local	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
164.gzip	0.94	0.93	0.89	0.92	0.91	0.96	0.97	0.96	0.97	0.97
175.vpr	0.86	0.9	0.94	0.94	0.92	0.86	0.91	0.95	0.96	0.95
176.gcc	0.89	0.91	0.91	0.93	0.94	0.89	0.93	0.93	0.95	0.96
181.mcf	0.94	0.87	0.98	0.96	0.91	0.94	0.88	0.99	0.99	0.97
186.crafty	0.91	0.89	0.91	0.93	0.92	0.92	0.91	0.94	0.95	0.96
197.parser	0.78	0.85	0.86	0.87	0.9	0.79	0.87	0.89	0.91	0.94
252.Eon	0.93	0.94	0.94	0.95	0.93	0.93	0.96	0.95	0.97	0.96
253.perlbmk	0.89	0.9	0.92	0.94	0.94	0.9	0.92	0.94	0.96	0.96
254.gap	0.92	0.94	0.94	0.94	0.95	0.93	0.95	0.96	0.97	0.97
255.vortex	0.94	0.96	0.96	0.95	0.97	0.94	0.96	0.96	0.96	0.97
256.bzip2	0.99	0.97	0.95	0.94	0.93	0.99	0.98	0.98	0.98	0.98
300.twolf	0.96	0.96	0.97	0.95	0.95	0.98	0.98	0.98	0.98	0.98
geomean	0.91	0.92	0.93	0.93	0.93	0.92	0.93	0.95	0.96	0.96

Figure 10. Average of Local Matching Scores

5.2. Local matching scores

Figure 10 shows the local matching scores. The local matching scores are reasonably high. On the average, about 91% of the top 10% of blocks in the maintained list remain in the top 10% of blocks in the accurate list. Benchmarks, such as 175,vpr and 197.parser, however, show relatively low local scores. For example, the compiler maintains only 86% and 78% of the top 10% blocks for the two benchmarks, respectively. In general, the weighted local matching scores are higher than their unweighted counterparts.

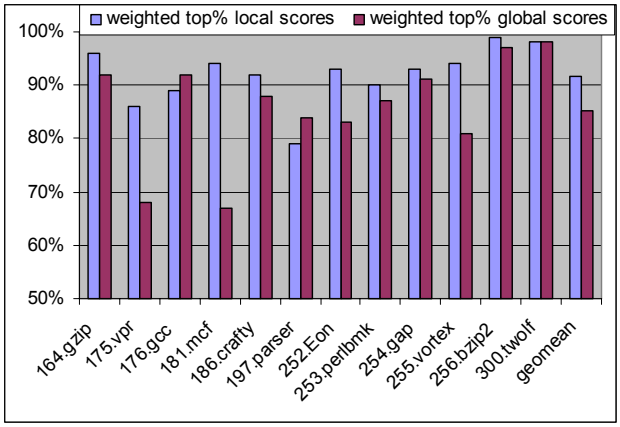


Figure 11. Comparison of Local and Glocal Matching Scores

5.3. Compare global and local matching scores

Figure 11 shows the weighted local and global matching scores for the top 10% of blocks. The local scores are higher than the global scores on the average. This indicates that the compiler maintains the relative ordering within individual functions better than the relative ordering across the whole program. However, a few benchmarks, such as 176.gcc and 197.parser, have lower local matching scores than their global matching scores. This is because the two benchmarks have complicated control flow graph with many switch statements. The complicated control flow graphs make the relative ordering of blocks within functions hard to maintain.

The benchmarks, such as 176.vpr and 181.mcf have very low global scores. Late we will show that the low global scores are caused by the function inlining optimization.

5.4. Distribution of functions by local scores

To take a closer look at the benchmarks, such as 197.parser and 176.gcc, with low local scores, we show weighted local matching scores for individual functions in the benchmarks. Notice that, functions with fewer than 10 blocks that are executed more than 500 times will not have matching scores and are not included.

Figure 12 shows the scatter graph of the functions by their weighted top 10% local matching scores for 197.parser. The matching scores often are either zeros, or between 50% and 100%. A noticeable number of functions (20 out of 120) have top 10% matching scores of zeros. For these functions, the top 10% of the blocks in the maintained list are totally different from the top 10% of blocks in the accurate list.

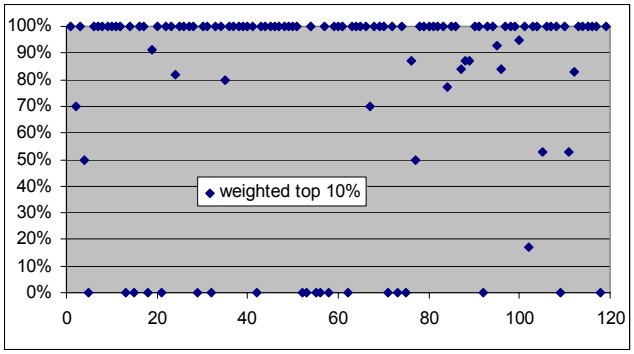


Figure 12. Local matching scores of functions in 197.parser.

Figure 13 shows the scatter graph of the functions for 176.gcc. The distributions are similar to that for 197.parse.

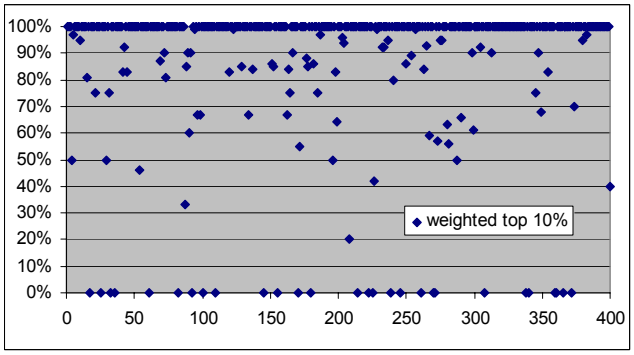


Figure 13. Local matching scores of functions in 176.gcc.

5.5. Effects of optimizations on matching scores

Benchmarks 175.vpr and 181.mcf have significantly lower global matching scores than other benchmarks. To determine the optimizations that may cause the low matching scores, we would need to instrument user programs after each optimization and compare the accurate profile collected at that time with the maintained profile. This is a difficult task, however, as adding a

profiling module after each of the optimization require significant amount of work.

In this experiment, we selectively turn off optimizations and compare the maintained profile with the accurate profile. If the matching scores increase when an optimization is turned off, we may deduce that the optimization caused loss of profile accuracy. We use the following compiler configurations for selectively turning off optimizations.

- *Base*: The default configuration for maximal performance.
- *NoI*: Turning off function inlining of the base configuration.
- *NoU*: Turning off loop unrolling of the base configuration.

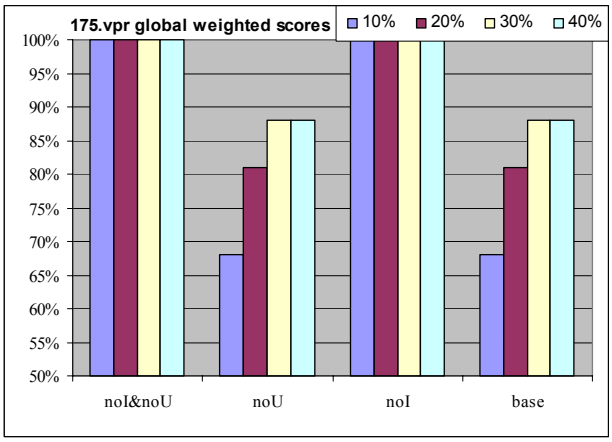


Figure 14. Optimizations and 175.vpr's global matching scores

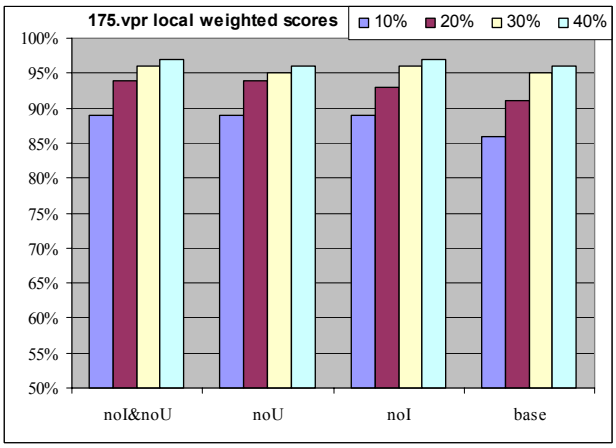


Figure 15. Optimizations and 175.vpr's local matching scores

The weighted global matching scores with different configurations for 175.vpr are shown in Figure 14. When function inlining and loop unrolling are both turned off (noU&noI), the maintained profile matches the accurate profile precisely. This indicates that function inlining or loop unrolling is responsible for the low global scores. Turning off loop unrolling (noU) and function inlining (noI) individually points out that function inlining is responsible for the low global scores for 175.vpr.

The weighted local matching scores with different configurations for 175.vpr are shown in Figure 15. Turning off loop unrolling (noU), function inlining (noI), or both (noU&noI) improve the local scores for 175.vpr only slightly.

The weighted global and local matching scores with different configurations for 181.mcf are shown in Figure 16 and Figure 17. Function inlining causes significant loss of global scores but only slight loss of local scores for 181.mcf.

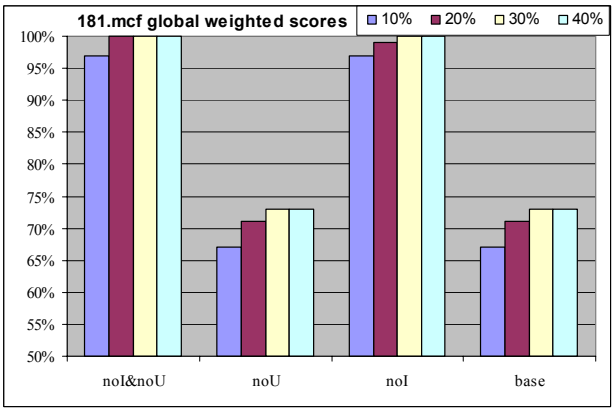


Figure 16. Optimizations and 181.mcf global matching scores

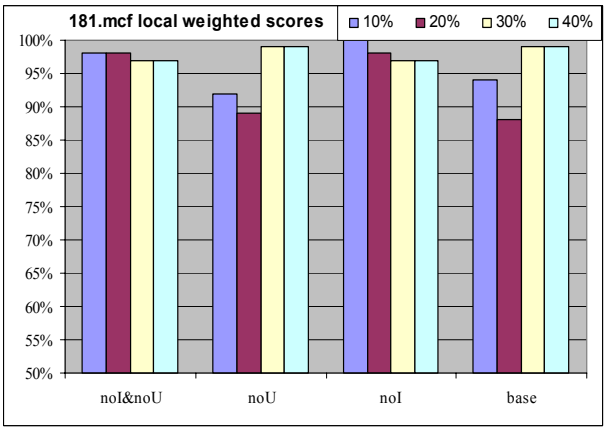


Figure 17. Optimizations and 181.mcf local matching scores

6. Conclusions and future work

In this paper, we study the accuracy of profile maintenance in an optimizing compiler. We measure the accuracy with the weighted and unweighted matching scores both locally and globally. In general, there is noticeable loss of profile accuracy both globally and locally, although the global matching scores are lower than the local matching scores. The loss of accuracy can be attributed to the fact that most of the profile maintenance operations follows the rule of probabilistic approximation. In particular, function inlining may be responsible for the global loss of profile accuracy.

The result in the paper is only the first step to investigate the accuracy of profile maintenance, which is an important subject in optimizing compilers for modern processors. This issue has not been considered before in literature and we hope our work will lead to more in depth investigations. The following problems remain open to be studied.

- Obtain more detailed understanding about optimizations and their affects on the loss of the profile accuracy. We briefly examined the effects of function inlining and loop unrolling on two benchmarks. More detailed study is needed to identify the optimization that causes most of the loss in profile accuracy.
- Enhance the profile information so it can be more accurately maintained. For example, we may collect additional information in the early profile phase, and use the information to guide the profile maintenance during optimizations. For the control flow graph in Figure 1 (a), the branch correlation information between blocks a, c, and e may suggest that the new frequencies in Figure 2 are more reasonable than those in Figure 1 (b).
- Design new profile maintenance methods that are easier to apply and more robust. One approach would be to log transformations rather than directly update the profile information after each optimization step. From the control flow graph before the transformation and the transformation log, a separate module may be designed to update the profile. This way, the profile maintenance is performed in one place and will be less error prone. Another way would be to only update profile in a few important places during optimization, and more subtle maintenance can be performed after the optimization.
- Order compiler optimizations to reduce the effect of profile inaccuracy on performance. It may be useful to apply the optimizations that will not lose profile accuracy earlier.
- Order compiler optimizations to tolerate inaccuracy in profile maintenance. It may be beneficial to apply the optimizations that need more accurate profile before the optimizations that may work with inaccurate profile.
- Qualify the performance impact due to loss of profile accuracy. Inaccuracy in a profile may not always lead to loss of performance. For example, a highly frequent block may not be optimized anyway due to the machine constraints, so misclassifying the block as infrequent should not affect the overall performance.

7. Acknowledgements

We would like to thank Yong-Fong Lee for his valuable comment about the paper, and our colleagues in Intel Programming Systems Research Lab and Intel Compiler Lab for implementing many of the profile maintenance operations. We appreciate the comments from the anonymous reviewers that helped improve the quality of the paper.

References

[1] Ball, T. and J. Larus, “Optimally profiling and tracing programs,” ACM Transactions on Programming Languages and Systems, 16(3): 1319-1360, July 1994.

[2] Ball, T. and P. Mataga, M. Sagiv, “Edge profiling versus path profiling,” Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages January 1998

[3] Bharadwaj, J., K. Menezes, C. McKinsey, “Wavefront scheduling,” Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture. November 1999.

[4] Calder, B., P. Feller, A. Eustace, “Value profiling,” Proceedings of the thirtieth annual IEEE/ACM international symposium on microarchitecture December 1997

[5] Chang, P. P, S. Mahlke, and W.M. Hwu, “Using profile information to assist classic code optimizations,” Software-practice and Experience, 1991.

[6] Chang, P. P, W.-W. Hwu, “Inline function expansion for compiling C programs,” Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation June 1989. ACM SIGPLAN Notices, Volume 24 Issue 7

[7] Gupta, R., Benson, D.A.; Fang, J.Z., "Path profile guided partial dead code elimination using predication," Proceedings, 1997 International Conference on Parallel Architectures and Compilation Techniques, 1997, Page(s): 102 -113

[8] Hwu, W.M., et al, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," The Journal of Supercomputing, Kluwer Academic Publishers, 1993, pp. 229-248

[9] Intel Corp, Intel® Itanium™ Processor Hardware Developer’s Manual, 2000. http://developer.intel.com/design/ia-64/manuals.htm.

[10] Kreahling, W. C., C. Norris, “Profile assisted register allocation,” Proceedings of the 2000 ACM symposium on Applied computing. March 2000

[11] Lavery, D. M., W.M. W. Hwu, “Unrolling-based optimizations for modulo scheduling,” Proceedings of the 28th annual international symposium on microarchitecture December 1995

[12] Mowry, T. C., C.K. Luk, “Predicting data cache misses in non-numeric applications through correlation profiling,” Proceedings of the thirtieth annual IEEE/ACM international symposium on microarchitecture December 1997

[13] Pettis, K., R. C. Hansen, “Profile guided code positioning,” ACM SIGPLAN Notices, Proceedings of the conference on Programming language design and implementation. June 1990

[14] Ramirez, A., L.A. Barroso, K. Gharachorloo, R. Cohn, J. Larribe-Pay, P. G. Lowney, and M. Valero, "Code Layout Optimizations for Transaction processing Workloads," ISCA28, June 2001.

[15] Sarkar, V., “Optimized unrolling of nested loops,” Proceedings of the 2000 international conference on Supercomputing. May 2000.

[16] Schlansker, M., S. Mahlke, R. Johnson, “Control CPR: a branch height reduction optimization for EPIC architectures,” Proceedings of the conference on Programming language design and implementation May 1999. ACM SIGPLAN '99 Volume 34 Issue 5

[17] Vahid, F., “Procedure cloning,” ACM Transactions on Design Automation of Electronic Systems (TODAES) January 1999, Volume 4 Issue 1

[18] Wall, D. W. "Predicting program behavior using real or estimated profiles," Proceedings of the conference on Programming language design and implementation, May 1991, ACM SIGPLAN Notices, Volume 26 Issue 6.

[19] Wu, Y., J. Larus, “Static Branch Frequency and Program Profile Analysis,” Proceedings of the 27th Annual ACM/IEEE international symposium on microarchitecture. November 1994.

[20] Young, C., M. D. Smith, “Better global scheduling using path profiles,” Proceedings of the 31st annual ACM/IEEE international symposium on microarchitecture. November 1998

Mastering Startup Costs in Assembler-Based Compiled Instruction-Set Simulation

Ronan Amicel
IRISA/INRIA
Campus de Beaulieu
35042 Rennes Cedex
FRANCE
ramicel@irisa.fr

François Bodin
IRISA/INRIA
Campus de Beaulieu
35042 Rennes Cedex
FRANCE
bodin@irisa.fr

November 30, 2001

Abstract

The increasing size and complexity of embedded software requires extremely fast instruction-set simulation. Compiled instruction-set simulation can provide high simulation speed, but the cost of generating and compiling the simulator can be a problem. We claim that efficient compiled instruction-set simulation with small startup costs is possible, using our assembler-level approach. We present ABSCISS, a retargetable and flexible system that generates optimized compiled simulators from assembler programs. Experimental results show the produced simulators to be significantly faster than interpretive simulators, and also show that our assembler-based approach allows to master the simulator generation and compilation times.

1 Introduction

Instruction-set simulation can be used to evaluate different instruction-set architectures (ISAs) in the context of architecture exploration, or to validate a compiler back-end, to test, tune and debug programs, on a user friendly PC or workstation rather than on actual silicon which might not even exist yet.

The increasing size and complexity of embedded software requires extremely fast instruction-set simulation. Compiled instruction-set simulation [8] is an approach that is potentially much faster than interpretation, but it has a startup cost due to the generation and compilation of the simulator. This startup cost is often seen as a major drawback and has limited the adoption of compiled instruction-set simulation.

In this paper, we present ABSCISS, a generator of compiled instruction-set simulators that works at the assembler level. We show that this approach allows to build a system that combines flexibility, accuracy and very fast simulation, along with a small startup cost.

ABSCISS automatically generates compiled simulators from a description of the target architecture. At present, this allows us to target various statically scheduled RISC and VLIW processors. Within this kind of architectures, the simulators generated by ABSCISS are cycle-accurate. That is, the simulator outputs the exact number of cycles needed by the target processor to run the program. Caches can be simulated by interfacing to an external module. Other architectures can be simulated at a functional level, that is only the behavior of the program will be simulated.

Compiled instruction-set simulation has some limitations, such as its inability to run programs containing self-modifying code or dynamically loaded libraries. Fortunately, these are seldom used in the context of embedded systems.

Unlike previous approaches [4, 5, 7, 10], the ABSCISS generator takes assembler programs as input instead of binaries. The machine description does not need to specify a binary encoding for the ISA, which, in the context of architecture exploration, enables faster prototyping. There is no need to specify the executable file format either. Besides, tools like an assembler or a linker for the target ISA are not required. We will also show how this approach helps reducing generation and compilation times.

To reach high performance, a compiled simulator has to be streamlined. That is, the generator should use as much static information as possible, so that the generated simulator only has to deal with the remaining dynamic part. ABSCISS includes a number of optimizations in the generator in order to produce an efficient simulator.

Moreover, if the user is only interested in a subset of the information that the simulator can produce, then only the necessary events should be simulated. ABSCISS works in a modular way, allowing it to produce a simulator tailored to the needs of the user, from a simple functional simulator to a detailed cycle-accurate one producing specific profiling information.

The rest of this paper is organized as follows. In Section 2, we discuss other works related to instruction-set simulation. In Section 3, we present ABSCISS, our simulator generator, along with the machine description used to retarget it. The overall performance of ABSCISS and the reduction of startup costs are discussed in Section 4.

2 Related Works

In this section we first present existing techniques for instruction-set simulation. Then we discuss automatic generation of functional and cycle-accurate simulators from machine descriptions.

2.1 Techniques for Instruction-Set Simulation

The basic technology for instruction-set simulation is interpretation. An interpreter uses a simple *"fetch-decode-execute"* loop, a technique that works with all kinds of programs, including self-modifying code. The major drawback is its poor performance, because each instruction has to be decoded over and over again. Attempts to make it faster involve caching instructions either in a predecoded form, or after translating them to the host instruction set, using dynamic binary translation.

Binary translation [12] converts a binary program from the target architecture to the host instruction-set. The translation can be either static, when the whole program is processed before execution, or dynamic when instructions are translated on-the-fly.

Compiled instruction-set simulation [8] is similar to static binary translation in that the whole target program is translated once. But instead of directly generating a binary, a compiled simulator generator produces a high-level language program implementing the target program's behavior. This program is then compiled using the host compiler. This makes compiled simulation independent from the host architecture, and allows to rely on the host compiler to perform low-level optimizations.

2.2 Automatic Generation of Instruction-Set Simulators

Most instruction-set simulators are bound to specific hosts and/or targets. While porting them to a small number of architectures might be reasonable, it is unpractical when fast and easy retargeting is needed. In that case, simulators can be automatically generated from architecture descriptions. An architecture description language (ADL) enables functional simulation by defining the behavior of each instruction of the target ISA. In addition, an ADL can describe

the structure of the processor, in order to perform cycle-accurate simulation. A more detailed survey of ADLs can be found in [13].

Interpretiv esimulators have been generated from descriptions written in languages such as nML [2, 7], ISDL [3, 4] or EXPRESSION [5]. While nML is not w ell suited to cycle-accurate simulation, ISDL currently allo ws to model VLIW arc hitectures with simple pipelines. EXPRESSION enables detailed modeling of the structure of both the processor and the memory hierarch y.

An approach was recently described [6], where both in terpretive and compiled simulators can be generated from an instruction-set model, extracted from a MIMOLA structural description.

The work that is closest to ours is the generation of compiled instruction-set simulators from binary programs [1, 9], using a machine description written in LISA [10]. LISA allows to model more complex pipelines than ISDL, but is limited to ASAP[1] scheduling.

3 The Absciss Simulator Generator

In this section, w e describe ABSCISS (*Assembly-Based System for Compiled Instruction-Set Simulation*), our generator of compiled instruction-set simulators. ABSCISS is based on the SALTO [11] framework, which supplies the parser for assembler programs, and provides an object-oriented in terface to their structure.

3.1 Machine Description

An instruction-set simulator must accurately simulate the functional behavior of a program. In addition, for a cycle-accurate simulator, the execution's timing also has to be simulated. How ever, even a functional simulator may need some

[1] As Soon As Possible.

timing information. For instance, on a VLIW processor the order of register writes depends on the respective latencies of the producing instructions.

In order to automatically generate simulators, ABSCISS extracts information about the target processor from an extended SALTO machine description. This description contains both behavioral and structural aspects of the target architecture. Additionally the description covers the syntax of the instruction-set, which allows the generator to parse assembler programs.

From our experience, it takes up to three months to write and debug a complete machine description. The cost of later changes to the instruction-set or to the reservation tables, for exploratory purposes, is then minimal. Moreover, the description is not limited to instruction-set simulation. It can also be used to retarget other tools, such as a code scheduler or an assembly-level code optimizer [11].

3.1.1 Structural Aspect

A SALTO description lists the processor resources, partitioned into storage resources and functional units. It then provides structural information in the form of *reservation tables* associated with instructions. A reservation table specifies which resources are used, read or written at each step of the instruction's execution. This structural information captures the static aspect of the execution's timing, which allows to describe statically scheduled processors, such as VLIW ones.

3.1.2 Functional Aspect

We expand SALTO machine descriptions by adding functional information that specifies the semantics of instructions. This information is written in a simple RTL-like language, based on types and operators defined in the Zephyr project [14], with a few extensions such as saturating arithmetics. Table 1

Instruction	Description	Semantics
`iadd`	*integer add*	`$3 = add( $1, $2 )`
`dspidualadd`	*dual clipped add of signed 16-bit halfwords*	`$3[0..15] = addss( $1[0..15], $2[0..15] );` `$3[16..31] = addss( $1[16..31], $2[16..31] )`

Table 1: Examples of instructions semantics.

shows examples of semantics specifications taken from our TriMedia machine description.

3.2 Simulator Generation

An assembler file is represented by SALTO as a set of procedures, each one being a control-flow graph where vertices are basic blocks and edges are branches. For each procedure in the target program, ABSCISS goes through all basic blocks and generates a tree-like representation of the semantics of instructions, using information from the machine description.

First, the semantic tree is checked for typing coherency, then it is simplified using optimization rules (see 3.3). Finally, C++ code simulating the behaviour of instructions is generated from the semantic representation. This is unlike some other approaches where the semantics of an instruction is expressed directly as C code, and where the generator textually substitutes the instruction's arguments.

We have implemented two different code generation strategies. The default strategy targets high performance, by using native C types and operators when the size of the manipulated values matches the host's native word size. When this is not possible, the generator falls back to a more flexible strategy. This backup strategy uses generic types and operators from a library, that allows to simulate target architectures with registers of any size.

When working at the symbolic assembler level, instructions do not have an address yet. Therefore, ABSCISS relies on the instruction labels in the control

flowgraph to simulate branches. When the target of a branch is determined to be in the same simulator function, a simple "`goto`" is generated. When the target is in another function, or cannot be statically determined, code is generated to use a two-level dynamic dispatch mechanism, that uses "`switch`" statements to bring control-flow to the right destination label.

This dispatch mechanism is also used to implement calls to external functions such as library or system calls. When such a call is made by the simulated program, the first-level dispatcher directs the control flow to a special function. This function emulates the call using the host operating system. It first retrieves the arguments from the simulated registers and/or memory, calls the host function, writes the result to the expected location, then gives control back to the program.

A drawback of the assembler-level approach, however, is the decreased accuracy of instruction cache simulation. ABSCISS needs to assign addresses to instructions by itself, possibly leading to a code layout different from the original.

3.3 Optimizations

A large part of the optimizations is done by the compilers. First, the simulated program is optimized by the target compiler, then the generated simulator may be optimized by the host compiler. By addressing optimization opportunities not taken in to account by the compilers, we aim at optimizing the simulation process.

3.3.1 Functional Simulation

To optimize functional simulation, ABSCISS simplifies the semantic tree by removing unnecessary computations introduced during its creation. To this end,

it uses static evaluation of operators combined with some algebraic simplifications. This ensures that the generated simulation code does not contain any inefficient computations.

3.3.2 Cycle-Accurate Simulation

To optimize cycle-accurate simulation on a VLIW architecture, the generator statically computes the total cycle count for each basic block using the structural information from the reservation tables. The cost in the produced simulator is then only one "add" instruction per simulated basic block, in addition to the possible cost of an external cache simulator.

3.3.3 Compilation Time

The compilation time of the generated simulator files depends on the size of the C function containing the simulation code. The compilation time increases in a super-linear way with the size of this function. Also, if this function is too big, some compilers may fail to compile it.

Thus, it is important that the generator outputs the source code for the simulator as a set of functions of manageable size. Therefore, we implemented an automatic splitting mechanism in ABSCISS. Instead of generating a single simulator function per input assembler file, the representation of the assembler file is split in to several slices. The cut points are chosen on basic block boundaries, according to a maximum number of bundles per slice.

The efficiency of this mechanism is evaluated in Section 4.2.

4 Performance Analysis

In this section we present the results of some performance tests. First, we compare the overall performance of ABSCISS with that of an industrial interpretive

simulator. Then, we show how the startup costs of compiled instruction-set simulation are mastered in our approach.

4.1 Overall Performance

We first compare the overall performance of our approach to a traditional interpretive simulator.

With interpretive simulation, the total simulation time depends on the program's dynamic instruction count. With compiled simulation, the total simulation time is split into generation, compilation and execution times. Generation and compilation times are a function of the static instruction count only, whereas execution time is a function of the dynamic instruction count.

4.1.1 Experimental Setup

Our experiments were performed using the TriMedia ISA, a VLIW architecture from Philips Semiconductors. Its main features are 128 general purpose registers, 5 issue slots per instruction, and RISC-like operations including floating-point and multimedia extensions.

The reference simulator was `tmsim`, the TriMedia instruction-set simulator distributed by Philips. `tmsim` is an interpretive simulator and takes binary programs as input.

We used an implementation of MPEG video decoding as our example target program, with input streams of various sizes. Both systems had to perform a cycle-accurate simulation, but assuming perfect caches. In the case of Absciss, we compared the effect of different optimization levels for the host compiler.

All the tests were run on a 440 MHz UltraSPARC-IIi workstation, and the host compiler was `gcc 2.95.2`.

4.1.2 Results

Input stream	Phase	tmsim	absciss			
			gcc -O0	gcc -O1	gcc -O2	gcc -O3
"Random" (38 KB)	*Generation*	0.0	173.5	173.5	173.5	173.5
	Compilation	0.0	162.3	188.3	332.6	333.6
	Execution	213.2	17.8	2.7	2.3	2.3
	T otal	**213.2**	**353.6**	**364.5**	**508.4**	**509.4**
"P enguin" (496 KB)	*Generation*	0.0	173.5	173.5	173.5	173.5
	Compilation	0.0	162.3	188.3	332.6	333.6
	Execution	643.5	52.4	8.8	7.6	7.5
	T otal	**643.5**	**388.2**	**370.6**	**513.7**	**514.6**
"Bear" (1432 KB)	*Generation*	0.0	173.5	173.5	173.5	173.5
	Compilation	0.0	162.3	188.3	332.6	333.6
	Execution	7205.8	617.2	92.4	77.4	77.1
	T otal	**7205.8**	**953.0**	**454.2**	**583.5**	**584.2**

Table 2: MPEG simulation times (in seconds).

The simulation times, for different input streams, appear in Table 2. With both simulators, the program gave identical results, and the computed cycle count was the same.

With a very small data set, the startup cost makes ABSCISS slower than `tmsim`. However, with the other data sets, the overhead is amortized thanks to the faster execution. The speedup of ABSCISS on execution ranges from 12x to 93x, depending on the level of compiler optimizations. When taking the startup cost into account, this leads to an overall speedup of up to 16x on these benchmarks. Higher overall speedups could however be achieved on longer running benchmarks.

The results show that our system has an initial overhead of 5 to 9 minutes in this case, due to the generation and compilation of the simulator. The generation time is fixed for a given program, and the compilation time varies with the optimization level. The preferrable optimization level depends on the execution time of the program. Except for the smallest one, the best tradeoff with our

data sets is `-O1`, however with larger data sets `-O3` might be better, since the longer compilation time would then be amortized by the faster execution.

To sum it up, we see that the generation and compilation overhead is quickly amortized on long simulation runs. It can also be amortized over multiple executions of the simulator with different input data, as this does not require recompiling. Thus, compiled instruction-set simulation significantly speeds up the simulation of large programs.

4.2 Startup Costs

In the classical approach to compiled instruction-set simulation, a simulator is generated from a binary program. ABSCISS, however, generates a simulator from a set of assembler files. A major advantage of this approach, is that the program is naturally split into multiple parts, making both generation and compilation faster.

To study the effect of the size of input programs on the simulator compilation time, we manually merged the dozen of C source files of our MPEG video decoder benchmark. We compiled the resulting C source file to a large TriMedia assembler file, which we then used as input for ABSCISS.

We then measured the total time needed to compile a simulator generated from our large assembler file. We used slice sizes from 128 bundles, resulting in 72 slices, up to 6144 bundles, resulting in 2 slices. Note that when we disabled the splitting machnism to get a single slice, the compiler failed to compile the generated simulator correctly. The compilation times are shown in Fig. 1.

These results show that the total compilation time highly depends on the size of the generated functions. By using a slice size of 512 bundles, the compilation time can be divided by a factor of 2.3 to 4.5, compared to a slice size of 6144. However, we see that smaller slice sizes are not desirable since they introduce

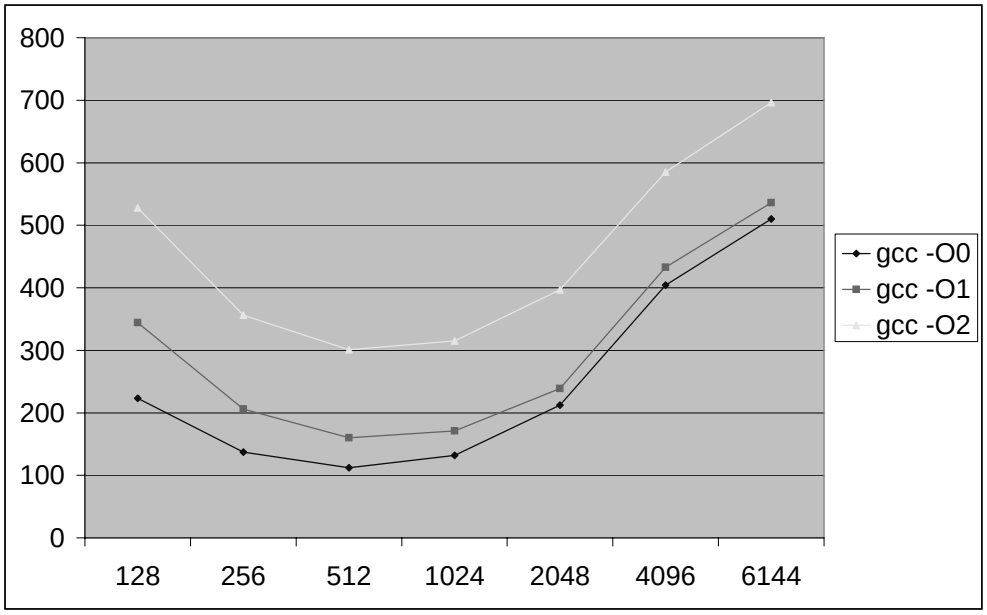


Figure 1: Total compilation time (in seconds) as a function of slice size (in number of bundles).

an additional overhead.

Thus, the approach used by ABSCISS to work at the assembler level leverages the natural division of a program into assembler files, allowing to master the generation and compilation times. Moreover, by automatically splitting large input files into slices of manageable size, ABSCISS can master the simulator compilation time even on large programs.

5 Conclusion

With embedded programs of increasing complexity, fast instruction-set simulation becomes a necessity. Compiled instruction-set simulation can deliver the simulation speed, but its associated startup cost can be seen as prohibitive.

In this paper we have presented ABSCISS, a generator of compiled

instruction-set simulators that aims at overall performance, by associating fast simulation and a mastered simulator compilation time.

From a set of assembler input files, ABSCISS automatically generates an optimized C++ program which simulates the behavior and the timing of the program. To this end, it uses a detailed machine description covering the semantics of instructions and their associated resource usage.

Our performance tests show that ABSCISS allows significantly faster simulation than a traditional interpretive approach, especially on long running programs. We also show that by working at the assembler level, and by adding a splitting mechanism, our approach helps reduce simulator generation and compilation times.

References

[1] Joachim Fitzner, Chris Schläger, Davorin Mista, and Vojin Zivojnovic. Implementing LISA Tools Based on a DSP Architecture Description. In *Proceedings of ICSPAT*, 1999.

[2] Markus Freericks. The nML Machine Description Formalism. Technical Report 91-15, TU Berlin, 1991.

[3] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 299–302, June 1997.

[4] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Proceedings of the* 36^{th} *Design Automation Conference*, pages 927–932, June 1999.

[5] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proceedings of Design Automation & Test in Europe*, March 1999.

[6] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators. In *ASP-DAC'99*, 1999.

[7] F. Löhr, Andreas Fauth, and Markus Freericks. SIGH/SIM - An Environment for Retargetable Instruction Set Simulation. Technical Report 93-43, TU Berlin, 1993.

[8] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled Instruction Set Simulation. *Software–Practice and Experience*, 21(8):877–889, August 1991.

[9] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language. In *Proceeding of Design Automation and Test in Europe*, 2000.

[10] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proceedings of the 36th Design Automation Conference*, pages 933–938, June 1999.

[11] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. SALTO: System for Assembly-Language Transformation and Optimization. In *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, pages 261–272, December 1996. (Also as IRISA Technical Report PI-1032 or INRIA Research Report RR-2980.).

[12] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary Translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[13] Hiroyuki Tomiyama, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. Architecture Description Languages for Systems-on-Chip Design. In *APCHDL'99*, October 1999.

[14] The Zephyr Compiler Infrastructure. `http://www.rcs.virginia.edu/zephyr/`.

On the Predictability of Program Behavior Using Different Input Data Sets

Wei Chung Hsu, Howard Chen, Pen Chung Yew
Department of Computer Science
University of Minnesota

Dong-Yuan Chen
Microprocessor Research Labs
Intel

Abstract

Smaller input data sets such as the test and the train input sets are commonly used in simulation to estimate the impact of architecture/micro-architecture features on the performance of SPEC benchmarks. They are also used for profile feedback compiler optimizations. In this paper, we examine the reliability of reduced input sets for performance simulation and profile feedback optimizations. We study the high level metrics such as IPC and procedure level profiles as well as lower level measurements such as execution paths exercised by various input sets on the SPEC2000int benchmark. Our study indicates that the test input sets are not suitable to be used for simulation because they do not have an execution profile similar to the reference input runs. The train data set is better than the test data sets at maintaining similar profiles to the reference input set. However, the observed execution paths leading to cache misses are very different between using the smaller input sets and the reference input sets. For current profile based optimizations, the differences in quality of profiles may not have a significant impact on performance, as tested on the Itanium processor with Intel compiler. However, we believe the impact of profile quality will be greater for more aggressive profile guided optimizations, such as cache prefetching.

1 Introduction

The SPEC benchmark suite [Henn2000] is a collection of CPU-intensive application programs. It has been widely used in the research community to evaluate architecture and micro-architecture designs and compiler optimizations. From SPEC89 to SPEC2000, the number of benchmarks as well as the execution time of each program has continuously been increased. The increase in execution time was most drastic when the SPEC92 was updated to the SPEC95. On average, each SPEC92int program executes about 1.3 billion instructions [Yung96] while this number increased to 64 billions for SPEC95int programs. In SPEC2000int, the average number of dynamic instructions executed becomes a few hundred billion instructions. With significantly increased execution time, and with more complex architecture/micro-architecture features to simulate, it becomes very difficult to simulate a complete run of the SPEC programs. A common practice in the research community is to take a small snapshot of the execution trace, for example, the first 100 to 500 million instructions of the trace. Alternatively, some researchers use smaller input data sets provided by SPEC to reduce simulation time[1]. In addition to the *reference* input sets, which give the complete run of each program, SPEC also provides the *test* data sets which give

[1] A survey of recent research publications shows that more than 60% of studies used reduced data sets.

a quick test of the benchmark, and the *train* data sets which allow the compiler to generate training profiles used for PBO (Profile-Based Optimization).

With the execution of a few hundred billion instructions in each program, the first 100 million instructions constitute about 0.1% of the total runtime, and are likely to perform initializations. Therefore, an initial snapshot of a simulation trace may be not representative. Even though some researchers wait until the initializations are complete, it is not clear how to obtain a snapshot that can represent the characteristics of the program. Several programs exhibit different execution phases, exercising completely different code and data behavior when it shifts from one phase to another. To accurately represent the execution of a benchmark program with multiple phases, at least one trace snapshot would need to be captured for each phase.

The approach of using reduced data sets may be more attractive, because the smaller input sets may exercise the similar execution phases like the reference input sets do. Since the test data sets and training data sets give a reduced runtime for the benchmark programs, a large amount of research has been conducted using these smaller data sets in their simulations to conduct faster performance evaluation. However, since the test and the train data sets were not originally designed to serve as reduced data sets for the reference input, they may exercise different execution paths in the programs than the reference input sets. If this is indeed the case, the performance evaluation conducted based on such input sets could be misleading. For example, if the complete run with reference input would cause significant I-cache misses and D-cache misses, but the run with test input incurs no cache misses, the evaluation results based on the test runs would be very misleading.

In 1992, Fisher and Freudenberge [Fish92] reported that branch instructions could be predicted statically by using previous runs of a program. This provides evidence to support Profile Based Optimizations (PBO). Starting in SPEC92, training input sets have been provided by SPEC for compilers to generate execution profiles and perform profile directed optimization. The success of using small data sets to predict branch directions for future runs may suggest that test or training input sets could be used to predict the program behavior for the reference runs. However, some recent studies [Cohn98] on post-link time optimizations report that an application may exercise different code when different users use the application. This observation is particularly common for general-purpose applications that are rich in features. Profiled based optimization has also advanced beyond static branch prediction. For example, some commercial compilers [Ayer98] have been using profiles to determine what procedures to optimize, what execution paths to get a high priority on resource allocation [Holl96], and which region to allocate more optimization time. Furthermore, recent research suggests using path profiling for trace cache

allocation [Rami99], using value profiling for value prediction optimization [Cald99], and using cache profiling for data layout optimization [Cald98]. It is therefore important to understand to what extent we may use one input data set to predict the program behavior of future runs.

In this paper, we evaluate how reliably we can use small input sets in place of more time consuming reference input sets. For some benchmark programs, small input sets exhibit the same execution behavior as the reference inputs, and the research community can comfortably use them to reduce simulation time. However, some programs do not have train or test input sets that are representative of their reference input set. We first examine the similarity of program behavior using high-level information such as execution profiles and IPC numbers. We then go into low-level analysis to investigate the frequent execution paths covered by each input data sets. Since the small and "light" input data sets generally do not stress the data cache as much as the reference input data set does, we also investigate whether different "heavy" input sets stress the data cache in a similar way. In other words, we would like to know how accurately and reliably we can use one input sets to predict the data cache behavior of a different set.

This study has two goals. One goal is to provide the research community some guidelines on using smaller input sets in reducing simulation time for SPEC benchmarks without giving misleading performance results. The second goal is to examine program behavior under different input sets. The key question is whether the smaller data set exercises the same execution paths and exhibits the same behavior as the reference input sets do? If not, we may not use the simulation results from smaller input sets to indicate the performance impact of the Spec2000 benchmark. Also, we evaluate the performance impact of using different input sets on the Itanium processor using the Intel compiler.

The remainder of this paper is organized as follows. In Section 2, we look at the high-level measures of execution profiles of Spec2000 programs using different input sets. In Section 3, we describe how to use the branch trace buffer feature in the Itanium processor to look into frequently executed paths exercised by different input data sets. Section 4 compares the frequent execution paths sampled by running different input data sets. Section 5 compares the execution paths for frequent data cache misses since many Spec2000int programs exhibit a high data cache miss rate. We evaluate the impact of different profiles on PBO performance in section 6, and the summary and conclusion are given in Section 7.

2 Execution Profiles and IPC Comparison

2.1 Execution Profile Comparison

We first examine the high-level performance characteristics of each benchmark program. This includes the gprof [Grah82] profiling and IPC information. In this study, we compile SPEC2000int benchmarks

Table 1. Execution time distribution of 181.mcf with different input sets

Procedure Name	Ref Input	Train Input	Test Input
price_out_impl	50.29%	31.06%	3.49%
refresh_potential	37.54%	39.24%	8.72%
primal_bea_mpp	8.47%	19.14%	54.65%
replace_weaker_arc	1.09%	1.99%	0.00%
sort_basket	0.76%	2.57%	18.02%

Table 2. Procedure coverage from one input set to the other

Gprof	Train vs Ref			Test vs Ref			Test vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Gzip	71.00	87.18	93.30	71.00	85.83	91.95	51.19	72.97	86.00
Vpr	29.65	73.61	83.58	55.19	78.20	86.06	59.75	84.55	92.31
GCC	64.45	84.86	91.04	66.15	85.24	90.82	55.89	76.91	86.99
MCF	37.64	88.06	96.56	8.49	9.26	97.74	19.30	21.89	92.79
Crafty	42.20	67.88	78.38	41.58	67.36	77.77	48.72	78.22	88.33
Parser	47.56	80.41	88.76	28.05	62.11	73.45	36.40	66.70	76.03
Eon	41.57	45.28	47.25	19.99	41.57	41.57	20.51	56.89	64.30
Perl	25.37	29.71	33.10	0.00	1.28	1.28	0.00	7.27	7.27
Gap	48.63	85.02	95.39	44.38	58.54	65.84	43.13	63.65	70.80
Vortex	33.86	53.56	68.87	37.85	65.80	71.58	48.50	73.50	87.99
Bzip2	5.57	49.29	59.03	27.27	53.45	67.93	24.56	40.79	93.34
Twolf	46.43	79.53	90.49	44.40	74.25	83.85	49.29	74.70	84.66

for a Pentium-III processor running on the Linux at O3 optimization level. Table 1 shows the execution time distribution from gprof of program 181.mcf. With the reference input set, the mcf benchmark spends 50% of time in procedure *price_out_impl*, 37.5% of time on procedure *refresh_potential*. When the train input is used, they are also the top two procedures in the profile. However, procedure *refresh_potential* now becomes the number one routine, while procedure *price_out_impl* reduces its execution time contribution from 50% to 31%. When the test input is used, the profile becomes very different. Now the top two procedures, *price_out_impl* and *refresh_potential* are insignificant, while procedure *primal_bea_mpp* and *sort_basket* become the top ones.

When a reduced input data set is used, we would like to know whether it covers the important part of the program for the reference runs. In Table 2, we try to correlate procedure profiles among different input data sets. For example, in the first column, we compare profiles of train input to the reference input. In the column labeled as 50%, we take the top procedures accounting for 50% of runtime cumulatively from the train input run, and give the percent of runtime these procedures cover in the reference input run. As shown in Table 2, Test input sets do not cover procedures very well for the reference run of Mcf, Eon, Perl, Gap, and Bzip. The procedures accounting for 80% of execution time of test input runs cover only 9.26%, 41.57%, 1.28%, 58.54%, and 53.45% of the reference run, respectively.

In general, train input runs have good procedure coverages. For Gzip, Vpr, Gcc, Mcf, Parser, Gap, and Twolf, the procedures accounting for 80% of execution time of train input runs cover similar execution percentages for reference input runs; in Perl, Eon, and Bzip the coverage is less than 50%. The compiler must be careful when training profiles are used to determine which procedures to optimize for these three programs. For example, Perl spends about 20% of time on procedure *regmatch*, but this procedure does not even show up in the gprof result for the training run. Therefore, using the training profile, the compiler might not optimize the *regmatch* procedure.

2.2 IPC comparison

In this section, we measure the IPC (Instruction Per Cycle) for each benchmark program using all three different input data sets. Several SPEC2000int programs, such as gzip, vpr, mcf, bzip and twolf, spend 90% of execution time on a very small number of procedures (less than 10), so the relative procedure coverage reported in the previous section is very high. However, it is not clear whether the execution behavior inside each procedure is similar under different input data sets. In this section, we examine the IPC numbers and in the next two sections, we sample the execution paths for a more detailed comparison.

We use hardware performance counters to report IPC numbers. The same set of benchmarks is compiled for Itanium using a beta version of the Electron compiler from Intel at O2 optimization level. For programs that have multiple reference input files, we average the IPC for each individual runs.

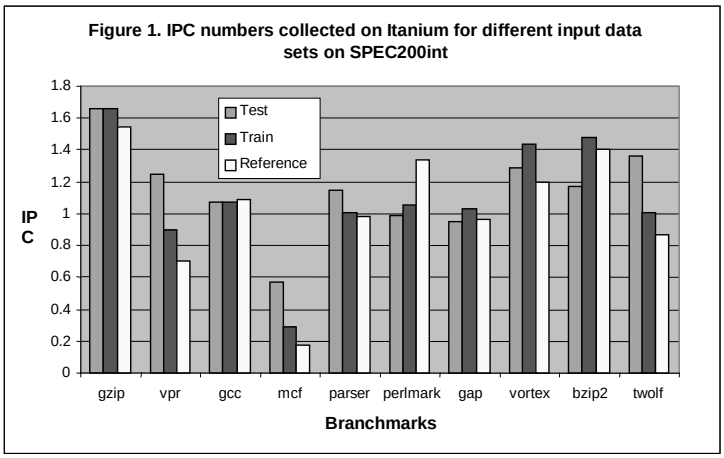


Figure 1 shows that the IPC numbers of vpr, mcf, perlmark, and twolf change significantly from one input set to the other. Consider vpr, for example, the IPC for the test run is about two times the IPC of the reference input. Mcf has an IPC number using test run more than three times the IPC using the reference input. In these benchmarks, different input sets exposed very different performance characteristics. For programs that have similar IPC numbers, such as gcc, gap, and gzip, it is not guaranteed that different input sets for such programs exercise the same execution paths and exhibit the same cache behavior. We will examine the sampled execution paths of each program to verify their behavior in following sections.

3 Using Branch Trace Buffer to examine frequently executed paths

The Itanium processor defines and supports a rich set of performance monitoring features that can be used to characterize workload and to profile application execution [Itan00a, Itan00b]. Itanium has four performance-counter registers that can be programmed to measure stall cycles in eight different categories

as well as to count occurrences of over a hundred events. Furthermore, Itanium supports Event Address Registers (EARs) for both Data and Instruction events as well as an eight-entry Branch Trace Buffer (BTB). The Instruction EAR can capture the addresses of instructions that trigger I-cache or ITLB misses. The Data EAR can capture the addresses of load instructions that cause D-cache or DTLB misses together with the target addresses of these loads. The Branch Trace Buffer is an 8-entry circular buffer that can capture information on the most recent branch instructions and their outcomes. These performance-monitoring features enable us to gain a detailed understanding of the dynamic execution behavior of the running application. Since each retired branch instruction that is recorded in the Branch Trace buffer may take up to two entries, one entry for the address of the branch instruction and another entry for the address of the branch target, we can program the Branch Trace Buffer to capture the most recent four taken branch instructions for each sample.

A profiling tool that utilizes the performance monitoring features, called Itanium Profiling Tool (or IPT), has been developed on Itanium running 64-bit Linux in the MRL of Intel. IPT required a customized performance monitor device driver (PMU driver) that runs as part of the Linux. IPT supports various modes of profiling on a running application, including the measurement of stall accounting, the counting of all performance events supported by the Itanium processors, and the collection of samples on various events. The IPT program interacts with the PMU driver to configure the performance monitoring registers and to receive profiling or sampling data from the PMU driver and store them in a profiling file.

To sample execution paths for this study, we used IPT to collect branch trace information for SPEC2000 integer benchmarks. We configure the Branch Trace Buffer to capture only the taken branches regardless of their branch prediction outcomes. While running the integer benchmarks with reference input, one branch trace sample was taken every one million cycles and every ten thousand D1 cache misses. For test and training input, one branch trace sample was taken for every ten thousands clock cycles and every one hundred D1 misses. This is because we try to maintain roughly the same number of samples between reference, test, and training runs. The sampling rates used is faster than the sampling rate usually used by gprof in all cases. Although a faster sample rate will obtain more unique execution paths, the most frequently sampled execution paths of each program remain the same as with the slower sampling rate. Since this study focus on the most frequently executed paths, we do not collect data on various sampling rates. Each branch trace sample was captured by the IPT program and stored to disk for offline processing. Offline, all branch trace samples were sorted to count the number of times each branch trace path was executed.

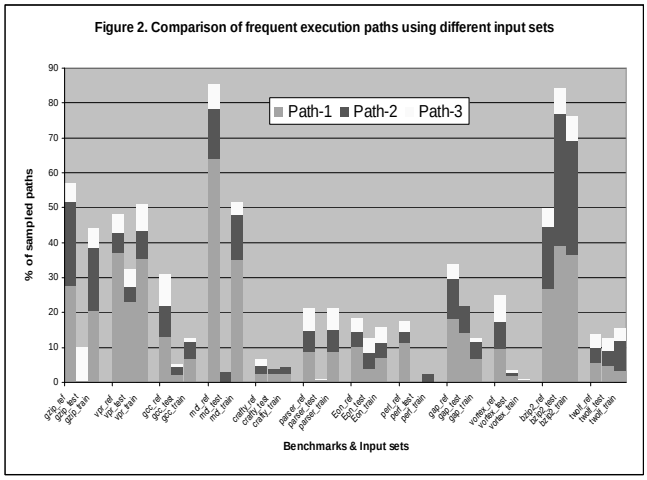


Table 3. Coverage of execution paths using one input run to predict the other

CPU	Train Vs Ref			Test Vs Ref			Test Vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Gzip	59.55	76.82	89.32	16.21	22.77	28.05	28.11	39.28	44.08
Vpr	48.01	64.89	69.89	47.01	80.80	92.06	65.70	91.10	96.09
GCC	69.69	85.99	91.32	76.65	88.70	91.59	64.46	84.08	88.77
MCF	66.16	91.72	96.59	2.83	18.79	18.86	9.11	26.18	26.69
Crafty	34.17	51.70	59.33	26.45	38.55	44.38	36.74	54.85	60.03
Parser	50.93	81.19	90.32	27.79	76.62	89.39	26.20	77.57	89.53
Eon	47.82	76.93	88.82	48.84	76.76	84.97	51.83	81.15	89.61
Perl	47.25	68.32	76.14	-	-	-	-	-	-
Gap	34.96	67.09	75.70	34.56	42.78	52.76	39.93	73.32	82.84
Vortex	41.40	80.61	89.03	63.21	84.17	91.66	44.99	75.00	87.03
Bzip2	26.99	52.24	72.07	26.99	44.59	56.58	36.41	69.16	82.02
Twolf	45.19	79.34	90.44	42.07	71.71	79.25	45.03	71.58	77.64

4 Execution Path Analysis for Different Input Sets

As we stated earlier, even if a program has similar procedure coverage and IPC numbers for different input data sets, the execution paths exercised by the different input sets may be different. If different execution paths are exercised under different input sets, using one input set may not reliably predict the performance of other runs. For the same reason, aggressive PBO based on one input set may not be effective for other runs.

We use the IPT tool described in Section 3 to study the frequent execution paths for each program under different input sets. We first select the top three frequently executed paths from the reference input runs. For each path, we report its percentage in the total sampled paths. For example, as shown in Figure 2, the number 1 path of Gzip accounts for 27.69% of total sampled paths. We also report their respective percentage for test and train input runs. The number 1 path selected from the reference run of Gzip accounts for only 0.26% from all the sampled paths for the test run, and 20.71% for the train input.

Figure 2 shows that the top three paths of Gcc using reference input account for about 30% of execution time. This seems to contradict with the common sense that Gcc tends to have a very flat profile. The hot execution paths come from the memcpy and the memset library routines. These two routines also account for 30% of execution time on both Pentium-III based and Sun Ultra SparcIIe based systems, using gprof with reference input.

In Figure 2, we can see that some important execution paths for the reference runs are insignificant for the train or the test runs. From the high-level comparisons in Section 2, we may believe Crafty, Gcc and Gap can reliably take advantage of reduced input data sets. However, Figure 2 shows that there are substantial variations on the relative importance of the frequently executed paths for these three programs.

Readers may wonder how important the relative order and their respective weight for those frequently executed paths are. For example, if for training input, path-1 accounts for 40%, and path-2 accounts for 10% of the execution time, and if the distribution becomes path-1, 10% and path-2, 40% for the reference input, would it make a big difference in PBO optimization? The answer depends on how the profile is actually used in optimization. If the optimizer uses the profiles to select all important paths, the relative order may not matter much. Eventually, both path-1 and path-2 will be selected and optimized. However, if PBO takes weight into account, it may decide to optimize only the number 1 path (due to compile time consideration), the outcome could be very different. Furthermore, when using small input sets for performance projection in simulations, the relative weight of different execution paths do make a difference.

The compiler may choose to take the top 80% of execution paths of the profile as optimization candidates. We are interested to know how much execution time these selected candidates may cover the run time of the full reference input run. Table 3 shows the possible coverage. For example, if we take the top 90% (accumulative) of execution paths from the profile collected with the Test input on Gzip, these paths may cover only 28.5% of the run time for the reference input run. In Table 3, we can see more than half of the benchmarks have very poor coverage if Test profile is used. In general, profiles using train input sets have better coverage than the test input sets.

5 Comparison of Frequent Execution Paths for Data Cache misses

5.1 Path coverage analysis

Figure 3 compares frequent paths leading to data cache misses. For programs without many data cache misses, it is not important to study such paths. However, since many SPEC2000int programs have a high D1-cache miss rate running on Itanium, it is important to understand whether such paths can be predicted using profiles generated from smaller input data sets. Figure 3 does not contain all the programs-- some programs with insufficient data cache miss samples were not included.

Figure 3 shows variations of such execution paths are far greater than the variations in Figure 2. For example, the path that accounts for the highest data cache misses in Vortex (responsible for 61.85% of D1-cache misses) does not even show up in the train input run (it covers 0.0% of sampled execution paths for data cache misses). Figure 3 shows that the test input is almost useless in predicting frequent D-cache miss paths except for Crafty. The train input can be used to predict data cache miss paths for Vpr and Parser. It may also capture frequent data cache miss paths for Gcc, Mcf, with substantially different weights on the paths. Train inputs predict data cache miss paths poorly for Gap, Vortex and Bzip2.

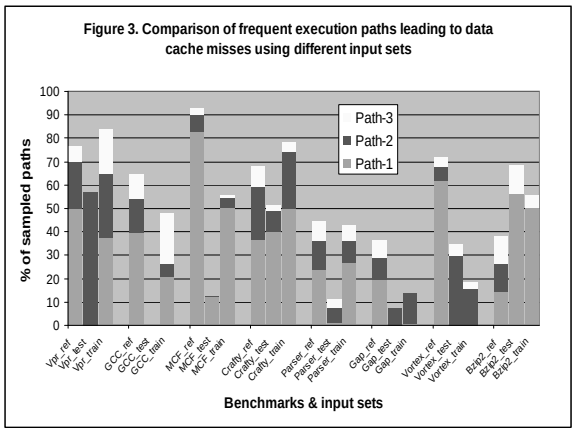


Table 4. Coverage of execution paths leading to data cache misses using one input run to predict the other

D1	Train Vs Ref			Test Vs Ref			Test Vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Vpr	37.02	76.93	78.17	20.23	20.23	21.47	27.85	27.85	35.92
GCC	15.06	73.92	92.12	0.00	0.00	0.00	0.02	0.06	0.07
MCF	82.70	85.33	93.51	2.18	85.44	86.07	25.83	84.12	88.29
Crafty	36.72	66.81	79.64	36.72	59.29	68.14	40.09	74.36	83.36
Parser	54.07	82.56	91.85	60.83	81.10	88.29	58.38	81.37	88.37
Gap	17.96	55.21	70.06	16.35	20.80	43.53	38.01	71.69	81.24
Vortex	16.95	25.37	28.61	13.84	24.18	30.88	35.07	71.84	83.78
Bzip2	14.44	31.86	51.97	14.44	29.97	35.84	50.21	65.77	73.25

Table 4 is similar to Table 3. The execution path coverage in Table 4 is in general lower than the coverage in Table 3. In particular, if the threshold is 50%, the prediction for reference run can be very poor. Table 4 shows test data sets can capture many frequent execution paths leading to data cache misses for Parser. However, from Figure 3, we have observed that the top three paths for data cache misses in Parser do not stand out during test data set runs. This shows a difference of using test data sets for PBO and for reducing simulation time. If the PBO compiler takes 90% of observed paths from one run to optimize for the other run, the relative weights of each path become less important. As long as the frequent executed paths are optimized, PBO has achieved its goal. However, the relative weights and order of such paths are important when simulation time is considered.

5.2 Small Vs large input data sets

From Table 4, we might conclude that for data cache profiling, using small input data sets may misrepresent the projected performance. It seems like the compiler should avoid using small input data sets to collect data cache miss profiles because reduced memory accesses are less likely to generate frequent data cache misses. However, the remaining questions are a) is it practical to use large input data sets to collect profiles for PBO and b) Even if a large input data set is used for profiling, can it reliably identify execution paths to the data cache misses for the other input set. For question (a), we shall leave it to software vendors to decide how much profiling overhead they can tolerate. For question (b), we looked at the predictability of using one reference input to predict for future runs with different input sets. Note that in this case both input data sets are from reference sets, not from the small data sets.

For those programs that incur frequent data cache misses on the Itanium and have multiple reference input files, we compare their most frequent paths leading to data cache misses in Figure 4. It shows that even if the full reference input is used to collect data cache miss path profiles, the variation is still large from one input to another. The number one execution path to data cache misses in Vortex account for 71.5% of all

sampled paths when the input lendian1.raw is used. However, this path does not appear when input lendian2.raw and lendian3.raw are used. On the other hand, a path that accounts for 16% of the sampled paths for input lendian2.raw and lendian3.raw, contributes only 4% when lendian1.raw is used. Similar results can be found for Bzip2 and Gcc. Using different input sets, no matter whether they are reduced size or regular size, may not reliably predict the paths leading to data cache misses for Spec2000int benchmarks.

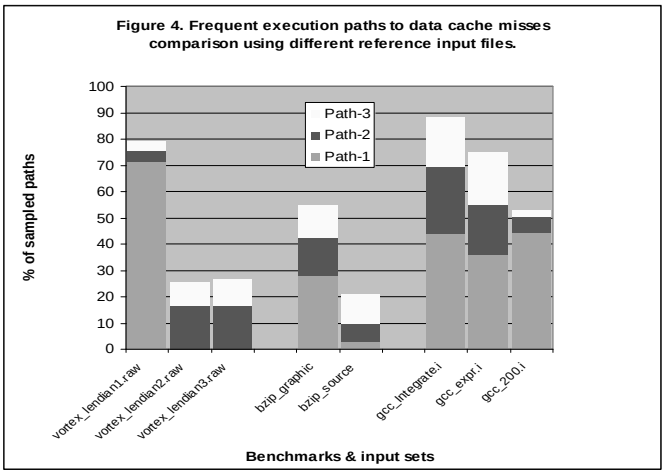


Table 5. Performance of PBO on Itanium using different profiles

Program	*IPO*	*IPO+PBO (test)*	*IPO+PBO (train)*	*IPO+PBO (reference)*
164.gzip	1.07	1.19	1.31	1.29
175.vpr	1.15	1.19	1.19	1.19
181.mcf	1.03	1.04	1.03	1.03
186.crafty	1.25	1.29	1.3	1.32
197.parser	1.08	1.11	1.11	1.11
254.gap	1.08	1.2	1.25	1.28
255.vortex	1.1	1.3	1.36	1.35
256.bzip	1.18	1.15	1.17	1.18
300.twolf	1.05	1.1	1.14	1.15
Average	1.11	1.17	1.21	1.21

6 Performance Impact of Different Profiles on Profile Based Optimization

Although our study indicates one input set does not always accurately predict the program behavior for another, it is not necessarily a problem in PBO because a) the compiler may select a set of inputs with different behaviors to generate profiles; b) some PBO transformations are less aggressive so that they depend less on the execution path or memory access behavior. In this section, we evaluate the performance impact of PBO based on profiles generated from the test, the train and the reference input sets. The experiment was conducted on the Itanium processor, where PBO is regarded as very important.

We compiled Spec2000int programs, using the Intel C/C++ compiler, on the RedHat 7.1 Linux. We used the performance of programs compiled at O2 as the base. We then compiled our benchmarks using IPO (Inter-Procedural Optimization) and PGO (Profile Guided Optimization, in Intel's term). Note that PGO is the same as PBO, so we call it PBO here. When compiled with IPO/PBO, we use profiles collected from test, train and reference input sets. The performance relative to the base performance is reported in

Table 5. All performance reported in Table 5 are relative to the base performance. As shown in Table 5, Gzip, Gap, Vortex, Bzip and Twolf can benefit from train profiles. This is no surprise; because Table 3 shows that train profiles cover the runtime of reference input better than test profiles on the aforementioned programs. Table 3 also shows that using profiles collected from reference inputs in PBO does not increase performance much. One thing worth noting is that the test profile of Mcf does not represent reference input at all. However, there is no performance difference for Mcf when more accurate profiles are used. This is because the performance of Mcf is dominated by several link-list chasing loops that have intensive data cache misses. Several existing effective PBO transformations would not improve those loops. However, if cache profile guided prefetching is implemented, using train profiles may expose such optimization opportunities.

Table 5 shows PBO can benefit from better profiles. The performance gain from using better profiles is not very significant, except for Gzip, which could gain 10% of performance if the train profile is used instead of the test profile. The performance impact of different profiles is not as significant as we expected. This is because Vpr, Mcf, Parser, Gap, Vortex and Bzip suffer significantly from frequent data cache misses on Itanium. When the performance is dominated by data cache misses, non-cache related PBO would not change performance much. When cache profile guided optimizations are adopted by compilers, different profiles would have a higher impact on performance.

7 Summary and Conclusion

It has been a common practice to use smaller input sets to estimate the performance of a benchmark or to generate profiles for PBO. In this paper, we look at how reliable this approach is. We have studied the high level metrics such as IPC and procedure level profiles and the low level measurement such as execution paths exercised by various input sets on SPEC2000int programs.

Our study indicates that the test input sets are not suitable to be used for simulation because they do not have an execution profile similar to the reference input runs. The train input is far better than the test data sets at maintaining similar profiles. However, there are significant differences between train profiles and reference profiles for Perl, Eon, Bzip2, and Vortex. We recommend cautiousness in using train input to project simulation performance for Vpr, Mcf, Gap, Gcc and Perl. We have observed significant variations in respective weights of those frequently executed paths using different input sets. Such relative weights could be critical when aggressive PBO is used. Profiles from train input could be reliable when predicting branch directions for other runs, but they could be misleading if the relative weights are used to guide optimizations.

A common practice has been adopted in PBO is to merge profiles from several different training input sets. However, most SPEC2000int programs have only one training input (only Perl and Eon have more than one training input files). In general, identifying representative small input sets for an application is not easy, even ISVs (Independent Software Vendors) have difficulties identifying representative sets. We have evaluated the impact of different profiles on PBO performance using the Itanium processor. While more accurate profiles lead to higher performance, the overall performance impact has not been shown to be very significant. Our study shows that smaller data sets do not predict frequent data cache miss paths in the reference input runs. We have also shown that data cache miss paths may not be predicted using a different reference input set. Since the profiled execution paths using small data sets often carry weights significantly different from paths in full runs, and since data cache miss paths are difficult to predict using different input sets, it would be a challenge to use profiles from small inputs to guide cache prefetching related optimizations.

8 References

[Ayer98] Andrew Ayers, Stuart deJong, John Peyton and Richard Schooler; "Scalable Cross-module Optimization", Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation, 1998.

[Burg97] Burger, D., and T. Austin, "The SimpleScalar Tool Set, Version 2.0" Technical report 1342, Computer Sciences Department, University of Wisconsin Madison, June 1997.

[Cald99] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", Jounal of Instruction Level Parallelism, March, 1999

[Cald98] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement", In Proceedings of the the 8th nternational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII) Oct. 1998.

[Cohn98] Robert S. Cohn, David W. Goodwin, P. Geoffrey Lowney, "Optimizing Alpha Executables on Windows NT with Spike", Digital Technical Journal, Vo l 9 No 4, June, 1998

[Fish92] Fisher J. A., and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct., 1992

[Grah82] Graham, S. L., Pb.B. Kessler, M.K. McKusick, "gprof: A Call Graph Execution Profiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, 1982.

[Henn2000] Henning, John L., "SPEC CPU2000: Measuring CPU Performance in the New Millennium," IEEE Computer, Vol. 33, No. 7, July 2000.

[Holl96] Anne M. Holler "Optimization for a Superscalar Out-of-Order Machine", Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, 1996.

[Itan00a] Intel Itanium Architecture Software Developer's Manual Vol. 4 rev.1.1: Itanium Processor Programmer's Guide. Intel Corp. July 2000.

[Itan00b] Intel Itanium Architecture Software Developer's Manual: Specification Update. Intel Corp. August 2001.

[Rami99] Alex Ramirez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero, "Software Trace Cache", 1999 ACM International Conference on Supercomputing (ICS), June 1999.

[Yung96] Yung Robert, "Design Decisions Influencing the UltraSPARC Instruction Fetch Architecture," Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, 1996

Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors

R. David Weldon, Steven S. Chang, Hong Wang
Gerolf Hoflehner, Perry Wang, Dan Lavery, and John Shen

Microarchitecture Research, Intel Labs
Intel Corporation
Santa Cara, CA 95054

Abstract

This paper examines the efficiency of the register stack engine (RSE) in the canonical Itanium architecture, and introduces novel optimization techniques to enhance the RSE performance. To minimize spills and fills of the physical register file, optimizations are applied to reduce internal fragmentation in statically allocated register stack frames. Through the use of dynamic register usage (DRU) and dead register value information (DVI), the processor can dynamically guide allocation and deallocation of register frames. Consequently, a speculatively allocated register frame with a dynamically determined frame size can be much smaller than the statically determined frame size, thus achieving minimum spills and fills. Using the register stack engine (RSE) in the canonical Itanium architecture as the baseline reference, we thoroughly study and gauge the tradeoffs of the RSE and the proposed optimizations using a set of SPEC2000int benchmarks built with differing compiler optimizations. Using a combination of frame allocation policies using the most frequent frame size, and using deallocation policies using dead register information proves to be highly effective. On average, a 71% improvement in RSE performance via a reduction of aggregate spills and fills can be achieved.

1. Introduction

The structure of register files and the organization of the memory subsystem is a fundamental trade-off in computer architecture. On one hand, memory latency still dominates the performance of many applications on modern processors, despite continued advances in caches. This problem, in fact, worsens as CPU clock speeds continue to advance more rapidly than memory access times, and as the data working sets and complexity of typical applications increase. Modern compilers incorporate powerful algorithms for memory disambiguation [1, 2] and register allocation [3, 4]. One trend in architecting modern microprocessors has been to expose a large register file architecturally and enlist the help of advanced compilers to map the working sets of the applications primarily into these registers, so as to reduce the number of load/store operations that can incur long memory access latency [5, 6, 7]. On the other hand, current trends in microprocessor design and technology lead to projections that the access time of a monolithic register file will become significantly higher than that of other common core operations, such as integer ALU operations [8]. To tackle this problem for traditional superscalar processors, various techniques have been proposed to reorganize the physical register file either through bank based partitioning or hierarchy based caching so that the access time to a small subset of registers can be made faster than access to the rest of the register file [8, 9, 7]. For these physical register file organizations, it is performance-critical to minimize spills and fills between the register files and memory.

Unlike most RISC architectures, the Itanium architecture [5] provides a general register file with 128 registers that are logically partitioned into 32 static registers, r0-r31, and 96 stacked registers, r32-r127. The static registers are globally visible to any procedure, but the stacked registers, with the exception of registers used for parameter passing, are only accessible locally within a procedure. Each procedure has its own variable sized

register stack frame. The architecture enables the compiler to explicitly specify the register stack frame size for each procedure. The Register Stack Engine is a processor state machine that maps a register stack frame onto the physical register file and copies (spills/fills) values to and from the register stack. Spilled register values are copied to memory, formally called the *backing store*. Fills occur when a register is used after having been spilled. The register value is then copied from the backing store into the register file.

This work is focused on the examining the relationship between RSE traffic (spills/fills) and the dynamic utilization of registers in the register frames. We first quantify the problem of internal fragmentation in allocated frames, and then propose a spectrum of optimizations that use dynamic information to improve the performance of the RSE. In essence, these optimizations transform the abstraction of local register frame management from a stack-based model into a much more flexible heap-based model, resembling traditional memory management mechanisms like garbage collection [10, 11].

Together with the baseline canonical RSE design, several RSE optimizations will be quantitatively evaluated using the workloads from a set of SPECint2000 Itanium binaries that are used for the official SPECint2000 rating in both *base* and *peak* performance categories [14, 15]. In addition, to help gauge RSE performance impact by both compiler optimizations and RSE optimizations, we also use a special set of *peak* binaries that are produced without loop unrolling optimization, so as to mimic binaries with lessened register pressure. We introduce techniques that perform dynamic register usage (DRU) based allocation, and deallocation based on *dead register information*, also commonly called *dead value information*, (DVI). For the benchmarks under consideration, the results show that these techniques can be highly effective in reducing the total register spills and fills by at least 30% over the baseline RSE. In particular, the best combination is to employ both most frequent use DRU based allocation and DVI based deallocation. On average, a 71% improvement in spill/fill reduction can be achieved over the baseline RSE.

The remainder of the paper is organized as follows. Section 2 reviews past research in register renaming for large register files. Section 3 presents background on Itanium RSE and illustrates with examples the potential problems associated with the static register frame management without dynamic usage feedback. Section 3 also provides motivation for optimizing the RSE. Section 4 explains a design space for optimizing the RSE using microarchitecture techniques and depicts 9 models with differing tradeoffs. Section 5 outlines the evaluation methodology and provides detailed performance analysis of these models in comparison to the canonical RSE. Section 6 concludes the paper.

2. Related Work

Register allocation, register renaming, and design issues regarding organization and management of large register files are closely related topics that have been extensively studied [8, 7, 9, 6, 17, 18, 19]. A detailed survey of these issues in modern processor designs can be found in [16].

Recently, Postiff et al [7] have proposed a register file caching scheme for implementing a large logical register file. Somewhat resembling the RSE in Itanium architecture, the design separates the logical register file from the physical register file and uses a modified form of register renaming to make the register cache easy to implement. The physical register file acts as a cache for the logical register file, which becomes the backing store. Some research [20, 27] also considered early deallocation of physical registers using dead register value information (DVI) to exploit the fact that after the last read of a register, that register becomes dead and can be reclaimed.

Unlike architectures considered in past research, the RSE in the Itanium architecture represents an infinitely large logical register file, which acts like a 96-entry cache for the top of the register stack. In the canonical Itanium processors such as the first generation Itanium products, the physical register stack file is implemented exactly as the architectural/virtual stack register file, and the RSE effectively guarantees a 1-to-1 correspondence between any logical register frame on top of the register stack and a physical register frame in the stacked registers. Both frames are of the same size, even if not all registers in a logical frame are used.

To our knowledge, this paper is the first work to thoroughly study the dynamic behavior of the Itanium RSE. In particular, the focus is on examining the dynamic usage of registers within the register frames that are explicitly managed by compiler (i.e. via alloc instructions) and quantifying the number of spills/fills incurred by the RSE. Furthermore, our work sheds light on significant benefits in using dynamic usage information to guide optimal register frame utilization. Finally, this paper introduces new insights on transforming the canonical stack-based RSE management model to a more flexible heap based management model, which enables more scalable physical implementations of the RSE in future Itanium processors.

3 Overview and Background on the RSE

3.1 Register Stack Frames

In the Itanium architecture, each procedure can have its own variable-sized (up to 96) register stack frame. A compiler's code generator uses the *alloc* instruction [5], to explicitly specify a procedure's register stack frame (Figure 1). The instruction allows for up to 8 incoming parameters (in), up to 8 outgoing parameters (out), the number of locally allocated stacked registers (local) and the number of rotating registers (rot) used in software-pipelined loops. The total number of registers in the register stack frame for a procedure is in+local+out <= 96.

The parameter passing registers for the caller and the callee (foo and bar in Figure 1) overlap. It is the task of the *register stack engine (*RSE*)* to map a register stack frame (*architectural registers*) to the stacked registers in the physical register file (*physical registers*). This mapping is transparent to the application software and the compiler. For example, in Figure 1, both procedures, foo() and bar(), access their first incoming parameter register as *r32*, but the RSE maps them to different physical registers, namely, *r32* and *r38*, respectively. Note: without loss of clarity, for simplicity in description, we will use the same notation to denote both *physical* and *architectural* registers.

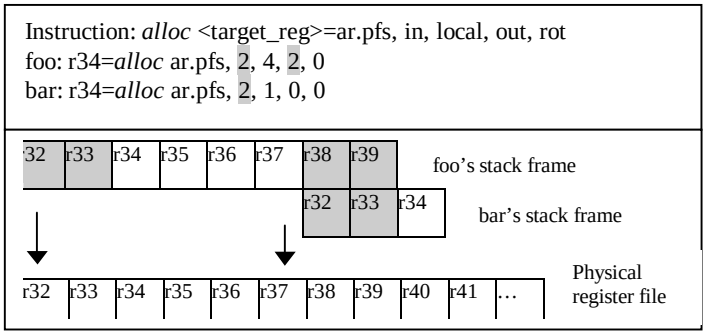


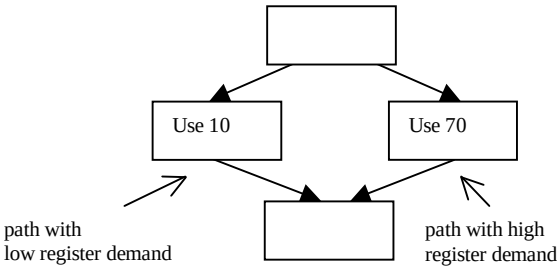
Figure 1. Alloc instruction and register stack frame

3.2 Register Spilling and Backing Store

The Itanium RSE manages the 96 architecturally visible stacked registers in the physical register file. From its point of the view, these stacked registers represent the top of the register stack and form a circular buffer. Upon entering a procedure call, when the callee allocates a new register stack frame, the RSE maps the frame (to ensure proper register renaming) to the register file. If the new frame does not fit, the RSE will spill registers allocated to a previous frame to the *backing store*, a memory area representing the logical register stack for the program. At procedure return, the RSE restores the caller's frame. If necessary, the RSE restores the registers by fills from the *backing store*. For example, assume four procedures *A*, *B*, *C* and *D* with *A* calling *B*, *B* calling *C*, and *C* calling *D*. If the register file does not have enough free registers available for the register stack frame of procedure *D*, the RSE spills registers starting with the stack frame of procedure *A*, to the *backing store*. When procedure *B* returns, the RSE will restore the original register stack frame of procedure *A*, filling A's registers from the *backing store* that were previously spilled.

Ultimately, register spilling is due to a deep call graph and/or inefficient utilization of allocated frames. One focus of this paper is to minimize internal fragmentation in allocated frames. To highlight the existence of internal fragmentation, two scenarios are presented below.

3.3 Imbalanced Paths



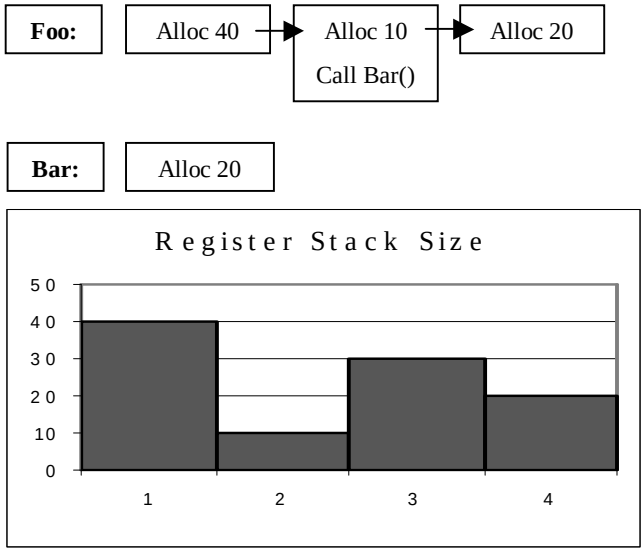
Example 1. Imbalanced Paths

Example shows two paths within a procedure with imbalanced register demands. Depending on the dynamic control flow, on one path, only 10 registers are needed, while on the other path, 70 are needed. Ideally, one should allocate 70 stack registers only when the path with high register demand is executed at run-time. Otherwise, only 10 registers should be allocated. If the path with low register demand is a frequently taken path, the dynamic register stack allocation could often avoid the 60-register internal fragmentation all together. This savings can thus help minimize register-memory traffic, as 60 registers would then be free for other procedures.

3.4. Nested Procedure Calls

Example 2 shows the potential to reduce the combined register stack size across procedure calls. Assuming the register stack frame size for foo is 40 registers and 30 of these registers are dead when foo calls bar(), at this point, only 10 registers are needed on the register stack.

Callee bar() could use the 30 dead registers of foo, instead of trying to find other free registers for a new frame. The histogram in Example 2 shows the potential dynamic register stack frame sizes for both procedures after stack frame sizes get dynamically adapted. Overall, no more than 40 registers are used. Without dynamic adaptation, foo() would allocate 40 registers, and bar() would allocate 20 additional registers for a total of 60. It is clear that without optimization, this behavior will quickly result in RSE spills.



Example 2. Shrinking the register stack before calls

3.5 Motivation of RSE Optimizations

Intuitively, in order to minimize RSE traffic, it is important to track and use dynamic frame sizes to allocate register frames so as to reduce internal fragmentation. In addition, it is important to relinquish dead registers as soon as possible so that they can be reused by the ensuing functions. To quantify the problem and seek effective solutions, we will evaluate a spectrum of RSE optimization schemes starting in the next section.

4 RSE Optimizations

To help define a set of RSE optimizations that employ different kinds of dynamic information, we first introduce two orthogonal types of dynamic information.

4.1 Dynamic Register Usage (DRU) *vs* Dead Register Information (DVI)

For each function, with respect to the static frame size specified in alloc, we track the total of registers that are actually used during the lifetime of the function's execution. This information represents the dynamic frame size, which is equal to or less than the static frame size. An allocation policy using this dynamic frame size information will be denoted as a *dynamic register usage* (DRU) policy.

A local register used in a function, is defined as "dead" after the last instruction that reads this register in this function is encountered. A dead register can be reclaimed and reused for future allocation and thus shrink the current register frame. A deallocation policy that can allow dead register relinquishment is called a *dead register information* policy. Such information is exactly the same as dead value information, i.e. DVI as discussed in [21]. Without loss of clarity, we will use dead register information and DVI interchangeably.

4.2 RSE Models

Table 1 summarizes the 9 models studied in this paper. Except for the baseline RSE (*Simple* model), all models use profile-based dynamic information on DRU or DVI, and non-overlapping frames between caller and callee is assumed. Therefore both frames effectively represent the parameter passing registers redundantly. The tradeoff between flexibility and redundancy will be further quantified as we cross-examine these optimizations.

4.2.1 Baseline RSE Model

Simple: This corresponds to the default Itanium RSE model that is implemented in recent Intel Itanium processor products, and thus is used as the baseline for this study. Frames are allocated as the alloc instruction is executed at the beginning of a function. The number of registers allocated is the same as the frame size specified in the alloc instruction. All registers in the frame are relinquished together upon return from the function. Parameter passing is achieved via the default mechanism where caller's output registers overlap callee's input registers.

4.2.2 RSE Models Exploiting DRU Only

Max Use: For every function, the dynamic register usage for each dynamic instance of this function is tracked, and the maximum number of registers used over all instances of this function is recorded as profile feedback. The code is then re-executed and the profiled

frame size for this function is always used to allocate register frame for each dynamic instance of this function.

Name	Use Alloc	DRU Size	DVI	When Free	Overlap Frame
Simple	Only	N/A	No	Return	Yes
Max	+Hint	Max	No	Return	No
MFU	+Hint	MFU	No	Return	No
Best Fit	+Hint	Exact	No	Return	No
Simple	Only	N/A	Yes	L+R	No
Max	+Hint	Max	Yes	L+R	No
MFU	+Hint	MFU	Yes	L+R	No
Best Fit	+Hint	Exact	Yes	L+R	No
Ideal	No	First	Yes	L+R	No

Table 1. RSE Optimizations: Shaded = DVI, L+R = Last use and on Return

Most-frequent-fit Use (MFU): For every function, for all of its dynamic instances, the most frequently recurring frame size is recorded as DRU profile feedback. The code is then re-executed. For each dynamic instance of this function, the profiled frame size is always used to allocate the dynamic frame. If more registers are required when the function is executed, additional registers will be reallocated on demand.

Best Fit*:* For each dynamic instance of a function, the total number of registers used by every dynamic instance of the function is recorded into a history. During the re-execution of the code, the history information is used as *oracle* knowledge to allocate the *exact* number of registers that will be used by any distinct dynamic instance of the function.

4.2.3 RSE Models Exploiting DVI only

The *Simple* DVI model is the baseline model without overlapping frames, and with profile based DVI information enabled. In other words, DVI information is used to deallocate dead registers just-in-time so as to allow the current frame to shrink in size even before the function returns.

4.2.4 RSE Models Exploiting both DRU and DVI

Max Use DVI, *Most-frequent-fit Use* DVI and *Best Fit* DVI correspond to the respective allocation policies described in section 4.2.2. DVI information is used in these RSE schemes to deallocate dead registers just-in-time so as to allow the current frame to shrink in size even before the function returns.

***Ideal* DVI**: for every register used in every dynamic instance of a function, information about this register's first use and last use are recorded as profile feedback. The code is then re-executed. Throughout execution of any dynamic function instance, a local register will not be allocated until its first use, thus growing the current register frame on demand. Similarly, once an instruction performs the last use of the register, this register is relinquished and the current frame is immediately shrunk. This register management model is demand-driven and resembles how physical registers are allocated and deallocated in the renamers of most out-of-order superscalar processors [16].

In this model, the entire physical register file becomes a heap that is used for allocation at individual register granularity. It is important to point out that *Best-fit*-DVI differs from *Ideal*-DVI in that, the former allocates all registers that will be used by the function at the beginning of the function call, while the latter starts the function with a 0-sized frame and only adds new registers onto the frame upon use of the register.

5. Experiments and Performance Evaluations

5.1 Simulation Environment and Workloads

To quantify tradeoffs of various RSE optimizations, detailed functional models of these RSE schemes are implemented in SMTSIM/IPFSim, a research, performance-modeling environment for Itanium family processors. It is adapted from the original SMTSIM [22] to Intel simulation infrastructure [23]. We use the number of spills and fills as the metric to gauge and compare effectiveness of RSE models. In an on-going work (though beyond the scope of this paper), we are using a cycle accurate processor model to further investigate the time cost of the RSE fills and spills and their impact to the performance of a given pipeline design in terms of execution time.

The workloads are selected from the SPEC2000int [14] and Olden suite [24]. The binaries are produced by Intel's production compiler Electron [12, 13] and they were used in the official SPECint2000 rating in both *base* and *peak* performance categories [15]. The SPEC2000int binaries are all evaluated for the first 2 billion instructions of execution. For Olden benchmarks, both *health* and *mst* are evaluated using the entire program execution, each with total instruction count at around 310 million instructions.

5.2 Performance Analysis

Figure 2 depicts the relative performance of all 9 RSE models. For each benchmark, the left chart compares the aggregate number of registers that are spilled/filled, while the right chart illustrates the number of occurrences of RSE fill/spill events. An event is defined as a necessity to fill or spill a set of registers from/to the backing store. All registers that are spilled will subsequently be filled, however the number of events will often differ. The event graphs indicate that in general, spill events are more common than fill events. This is due to the fact that a given stack frame can have its registers spilled from several other function calls before any of its registers are restored. Imagine a situation where function A calls B, calls C. C then spills 10 of A's registers. C returns and B calls D, which allocates 10 more than C, and spills another 10 of A's registers. D and B return and A's registers are restored (filled). C and D caused 2 spill events, where A only caused one fill event.

An In-depth analysis on tradeoffs for the RSE optimizations will be discussed in the rest of this section.

5.2.1 Max Use

Except for *gcc*, all the benchmarks profiled in Figure 2 exhibit worse *Max Use* performance than *Simple*. This behavior reflects the delicate tradeoffs associated with DRU policy. The inefficiency due to redundant representation of parameter passing registers in both caller frame and callee frame can limit gains from reducing internal fragmentation. We may see significant performance degradation when the number of output registers for a given function is large, causing more redundancy. As with all RSE models, the severity gets amplified as the call stack grows. *Max Use* performs well on *gcc*, because for several functions, the maximum frame size is much smaller than the static frame size. Therefore, the benefit of internal fragmentation overweighs the disadvantage of non-overlapping frames.

5.2.2 Most-frequent-fit Use (*MFU*)

Consistent performance improvement across all workloads seems to indicate that *MFU* is likely the best DRU policy. Not only does it outperform *Max Use* in every benchmark, it also outperforms the *Simple* model significantly on crafty, gap, gcc, and health. In fact, in the case of health, a benchmark that incurs frequent nested calls due to recursion, a 94% improvement can be achieved.

Furthermore, *MFU* outperforms *Best Fit* in all the benchmarks except mcf. Please see section 5.3 for an in-depth analysis of this phenomenon.

The use of *MFU* causes more RSE traffic than the *Simple* model on gzip, mcf, parser, and mst. Just as with *Max Use*, in these cases the gains made by reducing internal fragmentation did not outweigh the losses associated with non-overlapping frames. Nonetheless, *MFU* is a very attractive allocation policy as it is a history-based prediction. This implies that a simple value prediction scheme should suffice to perform such predictions effectively at run time.

5.2.3 Best Fit.

Best Fit goes a step farther than *Max Use* and *MFU* in that it allocates exactly the number of registers that will be used by each function instance. We therefore expect that in all cases *Best Fit* will produce fewer spills than *Max Use* and *MFU*. All benchmarks in the *Best Fit* model depict a significant decrease in total spills from *Max Use*. However as mentioned earlier, the only benchmark in which *Best Fit* outperforms *MFU* is *mcf*. See section 5.3 for more details.

Best Fit allocation will not always outperform the baseline *Simple* model, as shown by *gzip*, *mcf*, *parser*, *health*, and *mst*. In those scenarios, the benefit of usage-based allocation does not outweigh the disadvantage of non-overlapping frames.

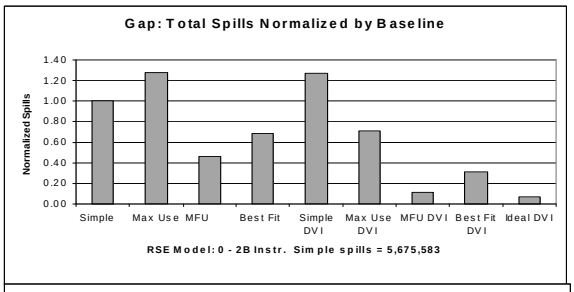
5.2.4 Simple DVI.

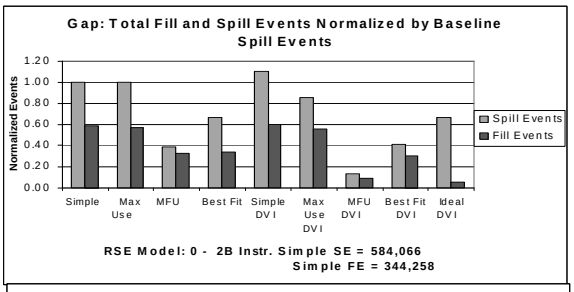
Simple DVI uses the baseline allocation scheme (static allocation based on frame size encoded in alloc instructions), and diminishes the current register frame size after each register's last use. As with all DVI models, we expect to see *Simple* DVI perform well against the non-DVI based models when the majority of the used registers have their last use close to the beginning of their respective functions. In particular, most input registers are used only once for parameter passing and right after the first use of these input parameters, the input registers are effectively dead and will be deallocated from the frame by DVI policy. So the redundancy incurred upon heap based frame allocation can be cut down drastically.

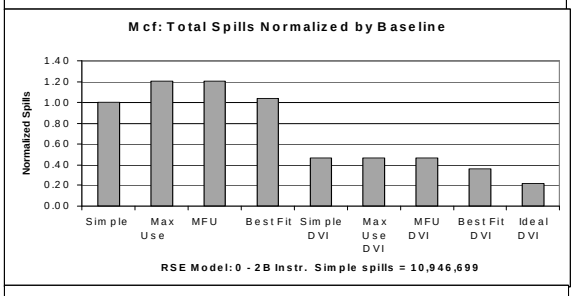
For *gzip*, *mcf*, *parser*, and *mst*, the DVI scheme shows a reduction in RSE traffic over the non-DVI models. The rest of the benchmarks show that *Simple* DVI does not outperform the non-DVI models. This indicates that the use DVI alone may not be a suitable solution to the fragmentation problem, and the conjunction of DVI and DRU should be closely examined.

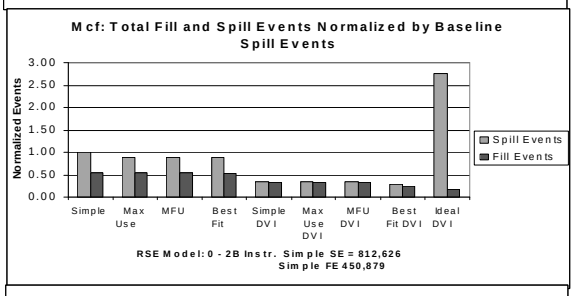
5.2.5 Max Use DVI.

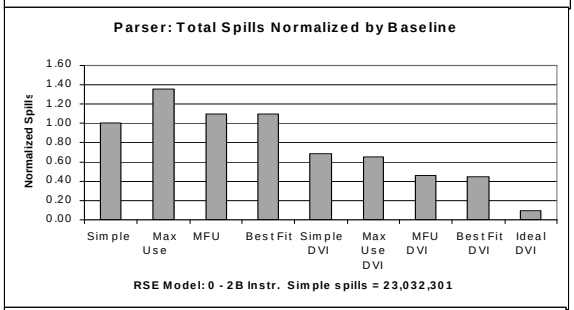
As stated before, *Max Use* is guaranteed to allocate frames of size less than or equal to that of the *Simple* model. Given that *Simple* DVI and *Max Use* DVI both have non-overlapping frames, *Max Use* DVI should

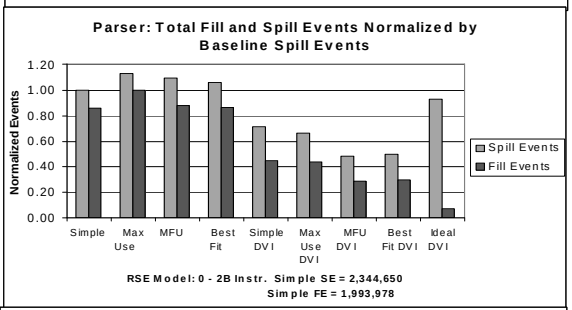

Gap: Total Spills Normalized by Baseline
Normalized Spills
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
RSE Model: 0 - 2B Instr. Simple spills = 5,675,583

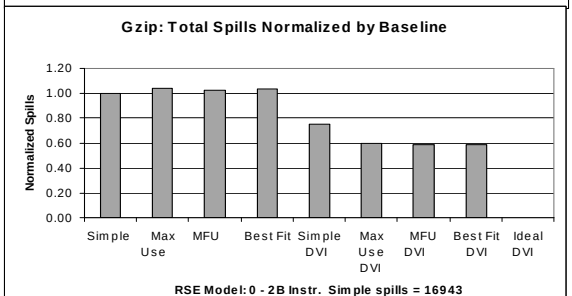

Gap: Total Fill and Spill Events Normalized by Baseline Spill Events
Normalized Events
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
Spill Events
Fill Events
RSE Model: 0 - 2B Instr. Simple SE = 584,066
Simple FE = 344,258

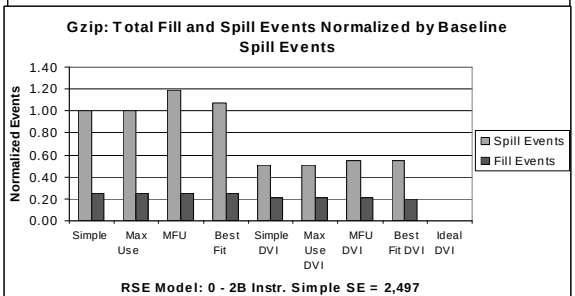

Mcf: Total Spills Normalized by Baseline
Normalized Spills
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
RSE Model: 0 - 2B Instr. Simple spills = 10,946,699

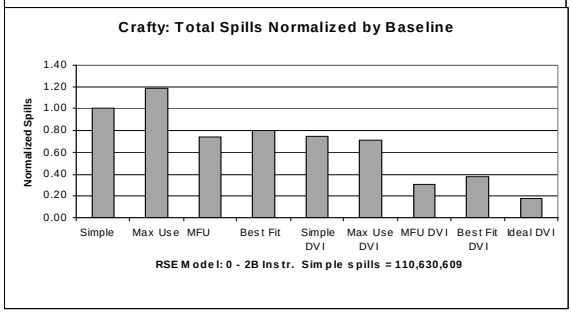

Mcf: Total Fill and Spill Events Normalized by Baseline Spill Events
Normalized Events
3.00
2.50
2.00
1.50
1.00
0.50
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
Spill Events
Fill Events
RSE Model: 0 - 2B Instr. Simple SE = 812,626
Simple FE 450,879

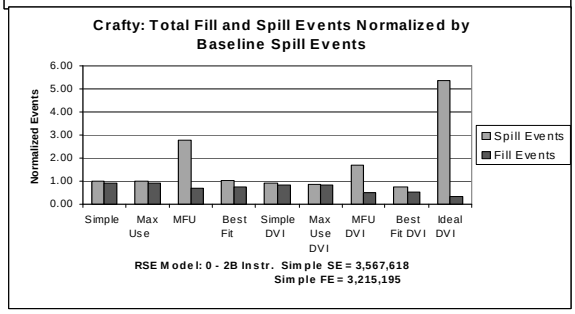

Parser: Total Spills Normalized by Baseline
Normalized Spills
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
RSE Model: 0 - 2B Instr. Simple spills = 23,032,301


Parser: Total Fill and Spill Events Normalized by Baseline Spill Events
Normalized Events
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
Spill Events
Fill Events
RSE Model: 0 - 2B Instr. Simple SE = 2,344,650
Simple FE = 1,993,978


Gzip: Total Spills Normalized by Baseline
Normalized Spills
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
RSE Model: 0 - 2B Instr. Simple spills = 16943


Gzip: Total Fill and Spill Events Normalized by Baseline Spill Events
Normalized Events
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
Spill Events
Fill Events
RSE Model: 0 - 2B Instr. Simple SE = 2,497
Simple FE = 630


Crafty: Total Spills Normalized by Baseline
Normalized Spills
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
RSE Model: 0 - 2B Instr. Simple spills = 110,630,609


Crafty: Total Fill and Spill Events Normalized by Baseline Spill Events
Normalized Events
6.00
5.00
4.00
3.00
2.00
1.00
0.00
Simple
Max Use
MFU
Best Fit
Simple DVI
Max Use DVI
MFU DVI
Best Fit DVI
Ideal DVI
Spill Events
Fill Events
RSE Model: 0 - 2B Instr. Simple SE = 3,567,618
Simple FE = 3,215,195

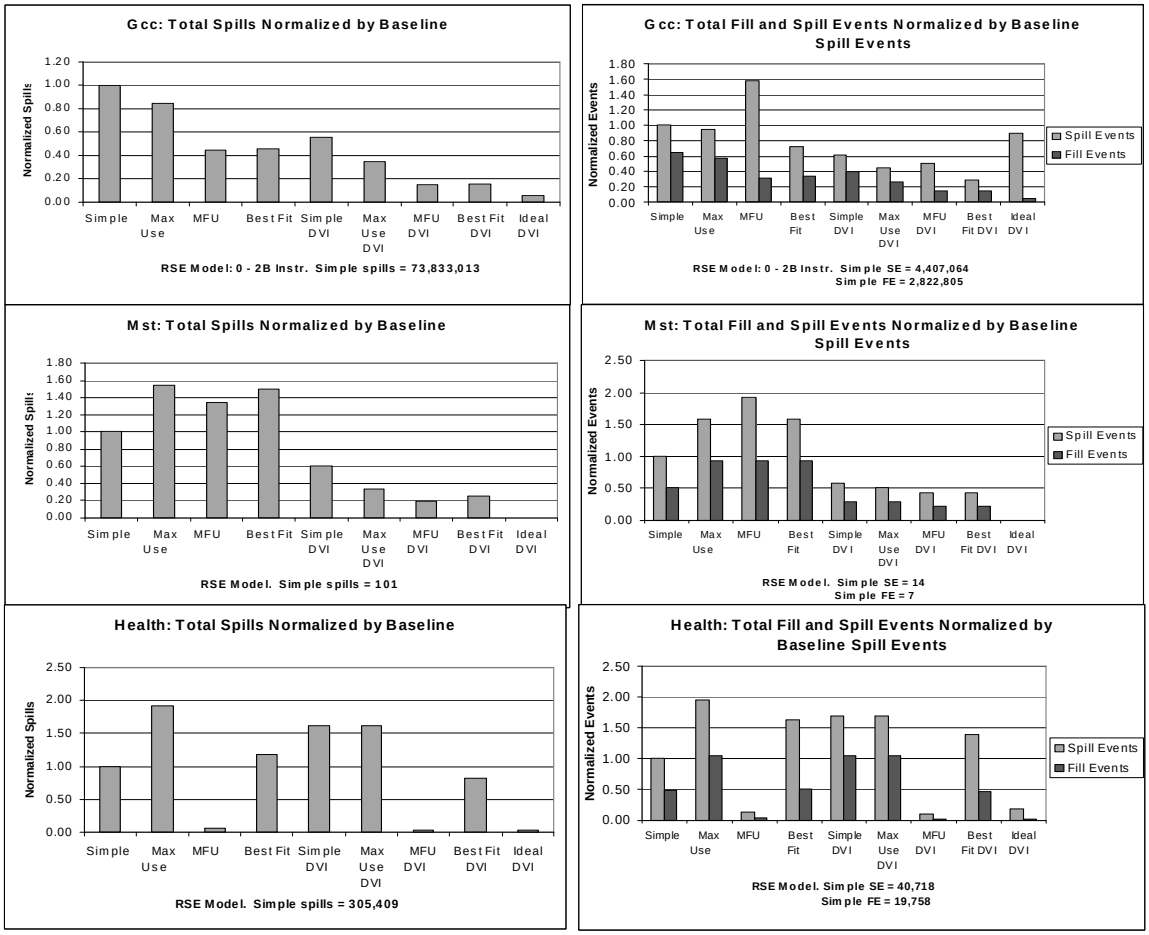


Figure 2: For each of 8 benchmarks, the left chart shows the normalized spills (in terms of register count); the right chart shows the spill/fill events (in terms of occurrence) for all RSE models.

clearly equal or outperform *Simple* DVI. *Max Use* DVI does outperform *Simple* DVI for all the benchmarks by an average of 20%.

5.2.6 Most-frequent-fit Use DVI.

Most-frequent-fit Use (MFU) is also guaranteed to allocate frames of size less than or equal to the *Simple* model and *Max Use* model. *MFU* DVI should be equal to or outperform *Simple* DVI and *Max Use* DVI. DVI model shows an average of 55% improvement over *Simple* DVI and a 46% improvement over *Max Use* DVI. For *gap* and *health*, the number of fills and spills in *MFU* DVI was an order of magnitude smaller than the *Simple* DVI and *Max Use* DVI models. With the exception of *mcf* and *parser*, *MFU* DVI also had fewer fills and spills than *Best Fit* DVI.

5.2.7 Best Fit DVI.

Similar to all the non-DVI models, we expect the *Best Fit* DVI model to produce fewer spills than *Simple* DVI and *Max Use* DVI. The average decrease in total spills from *Simple* DVI and *Max Use* to *Best Fit* DVI is 48% and 36% respectively. The comparison of *MFU* versus *Best Fit* is also the same for both non-DVI and DVI models. The *MFU* DVI model had fewer spills than the *Best Fit* DVI models for all benchmarks except *mcf* and *parser*.

5.2.8. Ideal DVI.

Ideal DVI defines the upper bound on optimal RSE performance. This allocation scheme uses oracle knowledge about the exact lifetime of every register, and it allocates and deallocates as necessary on a per-register basis. Therefore, at any point during the execution of a program, every frame is sized to exactly the number of registers in use. Not surprisingly, the data in Figure 2

supports the claim that *Ideal* DVI indeed outperforms all other proposed allocation schemes. *crafty*, *gap*, *gcc*, *mcf*, and *parser* all enjoy an order of magnitude reduction in the total number of spills when compared with the *Simple* DVI model. *health* has two orders of magnitude reduction in the number of spills compared to *Simple* DVI model. *gzip* and *mst* fit entirely in the register stack, and cause no spills using *Ideal* DVI.

5.3 In-depth Analysis of MFU

It seems counter intuitive that *MFU* can have fewer spills than *BestFit* for both the non-DVI and DVI models, since a *BestFit* policy assumes perfect knowledge on the exact frame size per every dynamic function instance. If the number of registers allocated by *MFU* is initially greater than the number of registers allocated by *BestFit*, then *BestFit* must perform as well as or better than *MFU*, depending upon whether DVI policy is used. Otherwise, it is possible for *MFU* to outperform *BestFit*. Figure 3 illustrates one such scenario where the number of registers initially allocated by *MFU* is fewer than that of *BestFit*.

In the call graph in Figure 3, we see that the most frequently used frame size for *A* is 6, the oracle knowledge in the *BestFit* model recognizes that the exact usage for the three instances of *A* are 12, 6 and 16 respectively. Consider allocation for function instance *i* in Figure 3 with *MFU* and with *Best Fit* in two scenarios: 1) *MFU* sees an under-allocation before the call to *B*, 2) *MFU* sees an under-allocation after the call to *B*. These are shown in Figure 4.

Example call graph

… A → B → C → A → C → D → A ….

BestFit : 12 6 16

Regs Alloc *i* *i+1* *i+2*

Instance #

… A → B → C → A → C → D → A ….

MFU Model:

Regs Alloc 6 6 6

Assume that 6 is the most frequent number of registers allocated for function A.

Figure 3. Best-Fit model allocates exact number of registers used for a given dynamic instances of function A. The MFU allocates the most frequent size for ALL dynamic instances of A.

In the first scenario, *MFU* allocates 6 registers upon entering *A*. Before the call to *B*, *MFU* detects under-allocation, and allocates 6 additional registers for *A*. This deferred allocation is no better than the *BestFit* case, which would allocate 12 registers upon entering *A*. Thus, *MFU* in this case does not outperform *BestFit*.

In the second scenario, *MFU* again allocates 6 registers upon entering *A*. *A* won't encounter the need for any additional registers until after *B* has been called and has returned. *B* now has the use of 6 additional registers. Based on our previous assertion that minimizing a given frame's size, minimizes future RSE traffic, it is clear that in this case *MFU* will outperform *Best fit*. This scenerio of under-allocation explains said performance in Fugure 2. This differred allocation is, in a sense, an approximation of *IdealDVI*, wich always minimizes the size of every frame on the stack.

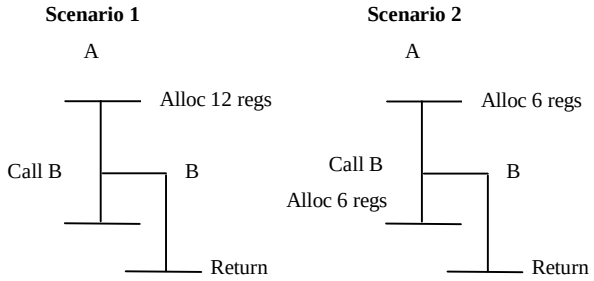


Figure 4. (1) MFU performance is equal to BestFit, (2) MFU outperforms BestFit

5.4 Performance of RSE Optimizations for Code with Differing Compiler Optimizations

The performance of the *Simple* policy is directly affected by the compiler which encodes the frame size in the alloc instruction. The results shown so far have been collected using the *peak* build binary and are used to primarily compare different microarchitecture specific RSE optimizations. It is interesting to gauge the effectiveness of these RSE optimization schemes in context of different compiler optimizations. In particular, we would like to find out whether the same relative merits between different RSE schemes manifest for *base* build binaries as well. Additionally, in order to measure the impact of code with lessened register pressure, we also disable the loop unrolling optimization in the *peak* build environment and use the resulting binaries, denoted as *peak_no_lur* to evaluate RSE schemes.

5.4.1 *Peak* Binaries vs. *Base* Binaries

In comparison between *peak* and *base* binaries, the *peak* binaries often have fewer spills than the corresponding *base* binary. In the case of *mcf*, there is an improvement greater than 250% in the reduction of spills for *Max Use* DVI and *MFU* DVI. *crafty*, *gcc*, and *parser* also incur fewer spills. However for *gap* and *gzip*, the *peak* benchmarks incur more spills than *base* binaries. *gap*'s *peak* results are slightly worse than the *base* binary. However, the *peak* for *gzip* is dramatically worse than its *base*. With the exception of the *Simple* model (which has overlapping frames), for *gzip*, all other RSE models using the *peak* binaries incur *two orders of magnitude more* spills than the *base* binaries. One likely reason could be that the loop un-rolling optimization enabled by default for the *peak* binaries greatly increases the register usage and lifetime of registers of interest. For a heap based allocation scheme, this would greatly exacerbate the impact of redundancy of input and output registers in the related frames. Consequently, the *peak* binaries would result in excessive spills.

5.4.2 *Peak* Binaries vs. *Peak_no_lur* Binaries

Even though the *peak* binaries do not demonstrate cross-board performance improvement in RSE schemes for all the benchmarks, the data does firmly indicate that differing compiler optimizations could have significant role in enhancing performance for the RSE optimizations.

The data indicates *peak_no_lur* binaries have fewer spills than the *base* binaries for all the benchmarks. By turning off the loop un-rolling feature, the compiler uses fewer registers for loops in *peak_no_lur* binaries than in *peak* binaries. The reduction in register usage directly translates to reduction in the total number of spills. In general, differing RSE schemes across the board, the relative performance of *peak_no_lur* binaries follow similar trends as that for the *peak* binaries. For both *gap* and *gzip*, the *peak_no_lur* binaries outperform both the *base* and *peak* binaries. In *gzip*, the *peak_no_lur* binary no longer incurs the excessive spills in its *peak* binary. Turning off the loop un-rolling feature significantly improved the RSE performance of *gzip*, across the spectrum of RSE designs.

5.5 Performance of RSE Optimizations For Varied Register Stack Sizes

As microprocessors approach higher clock frequencies, the ability to access the register file in one clock cycle becomes much more difficult, especially with the large register file sizes on the current Itanium processor families. It is of great interest to examine if the RSE optimization with DRU and DVI may allow us to get the same performance with a smaller register file. The same RSE simulation and analysis is performed using one CPU2000 base binary (*mcf*) and one Olden binary (*health*), with register stack sizes ranging from 64 to 128. The maximal dynamic frame sizes among all functions for both benchmarks are below 64, thus suitable for our analysis. The results are shown in Figure 5. All data are normalized to that for stack size of 96, the baseline RSE stacked register size.

For both benchmarks, the total number of spills increases as the register stack size decreases from 128 registers to 64 registers, which is expected. For *mcf*, the average increase in total spills is 40% when the stack size decreases from 96 to 80 registers. The average increase in total spills is 105% when the register stack size shrinks from 96 to 64. For *health*, the average increase in total spills when the register stack decreases from 96 to 80 registers and 96 to 64 registers are even worse, at 125% and 537% respectively.

Notice, however, that the difference in spills between the non-DVI models and its respective DVI models increases as the size of the register stack decreases. The DVI models do spill more with smaller register stack sizes but not as much as the non-DVI models. This highlights the relatively more significant role that DVI can play as register file sizes shrink: DVI clearly helps ensure the scalability of a RSE. In the case of *mcf*, all DVI-models outperform their non-DVI counterparts by having approximately 50% fewer spills. With the exception of the *Simple* model in *health*, all of *health*'s DVI models outperform their non-DVI counterparts. For *health*, the reason that *Simple* outperforms *Simple*_DVI is due to the redundancy existent in the non-overlapping frames in the DVI models. It is worth noting that for *health*, both *MFU* and *MFU* DVI have an order of magnitude fewer spills than the other models (with the exception of Ideal_DVI) for all register stack sizes. This echoes the previous observation that the *MFU* DRU policy can be a very effective allocation policy.

The combined use of DRU and DVI as shown in the *MFU* DVI model will enhance the RSE's performance significantly. For *mcf*, the *MFU* DVI model with 64 registers is equivalent to the *Simple* model at 96 registers. For *health*, the *MFU* DVI model at 64 registers outperforms the *Simple* model at 112 registers.

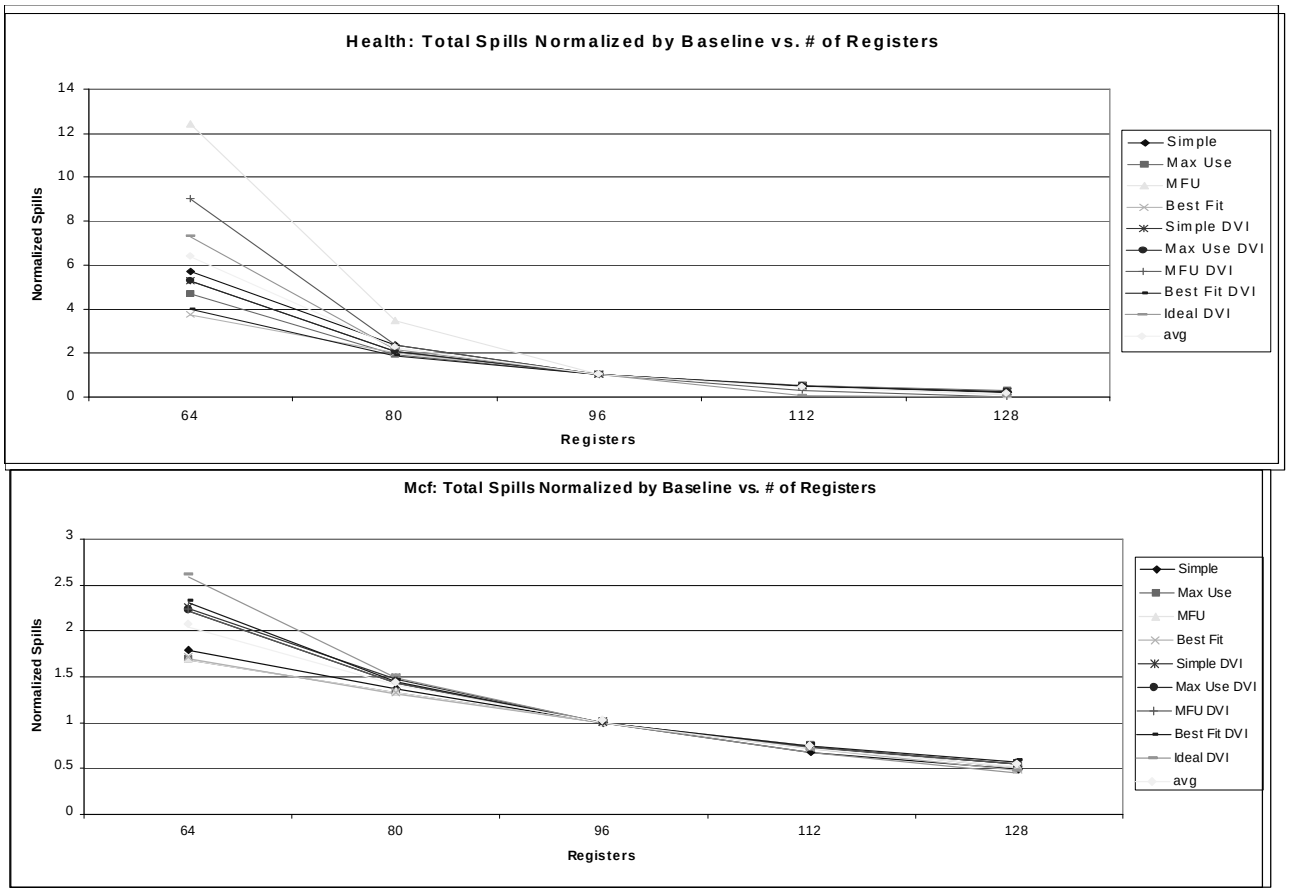


Figure 5: **The total number of spills made by the RSE models with various register stack sizes: (a) Mcf, (b) Health.**

5.6 RSE Performance with Varied Register Stack Sizes and Compiler Optimizations

In this section, we further investigate impact of varied register stack sizes to binaries built with different compiler optimizations across all RSE optimizations. To this end, we choose *mcf*. Its *peak*, and *peak_no_lur* binaries are evaluated and compared for all RSE models with varied register file sizes.

The *peak* and *peak_no_lur* binaries for *mcf* behave rather similarly. As stated earlier, both *peak* and *peak_no_lur* had significantly fewer spills than the *base* binary. This continues to be the case for all the register stack sizes. With a stack size of 64 registers, *MFU* DVI of *peak* and *peak_no_lur* have about 45% fewer spills than the baseline binary.

6. Conclusion

The common goal of the set of RSE optimizations in this paper is to minimize potential inefficiency of register file utilization for the stacked registers. To achieve this goal, the key learning from this research can be summarized into three axioms which can help guide optimal RSE implementations in future Itanium processors.

1. **Trim internal fragmentation**. Since compiler-determined static frame size tends to represent the worst-case register usage scenario that rarely occurs, it is essential to use dynamic usage information to allocate frame sizes close to the number of registers that will actually be used. This can lead to a drastic reduction of the internal fragmentation problem. The benefits are clearly demonstrated by RSE organizations using DRU based allocation policies.

2. **Timely (or sometimes lazy) allocation of what's needed**. For registers within a given allocated frame, the life times for individual registers are NOT necessarily persistent across the entire lifetime of the corresponding function, so it is beneficial to do partial allocation in a lazy fashion. The performance advantage of a *MFU* policy over a *Best Fit* policy, albeit counter-intuitive upon first sight, highlights the advantage of deferred partial allocation.

3. **Timely deallocation of what's no longer needed**. The performance advantage of RSE schemes that use DVI based deallocation policies is pronounced. Overall, the combination of an allocation policy using most frequent frame size and deallocation policy using dead register information proves to be highly effective and can achieve on average 71% improvement in reducing aggregate spills and fills over the canonical RSE.

References

[1] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In SIGPLAN Conference on PLDI, pp. 47-58, June 2001

[2] Robert P. Wilson and Monica S. Lam. Efficient Context-sensitive pointer analysis for C programs. In SIGPLAN Conference on PLDI'95.

[3] C.J. Chaitin et al. Register allocation via coloring. Computer Languages, vol 6. no. 1, pp 47-57. 1981

[4] F. C. Chow and J. L. Hennessy. The priority-based coloring approaches to register allocation. ACM Trans. Programming Languages and Systems. Vol 12. No. 4. pp. 501-536. 1990.

[5] Intel Corp. Intel IA-64 Architecture Software Developer's Manual.

[6] Matthew Postiff. Compiler and Microarchitecture Mechanisnms for Exploiting Registers to Improve Memory Performance. Ph.D. Thesis. University of Michigan, March 2001.

[7] Matthew Postiff, David Greene, Steve Raasch, and Trevor Mudge. Integrating Superscalar Processor Components to Implement Register Caching. Proc. 15th Intl. Conf. on Supercomputing, June 18-21, 2001. Sorrento Italy.

[8] Jose-Lorenzo Cruz, A. Gonzalez, M. Valero and N. P. Topham. Multiple-banked Register File Architectures. Proc. 27th ISCA, pp 316-325, June 2000

[9] Robert Yung and N. Wilhelm. Caching Processor General Registers. ICCD'95. pp. 307-312. Oct. 1995

[10] Richard Jones. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, Ltd, ISBN 0-471-94148-4, 1999.

[11] Dirk Grunwald and Benjamin Zorn Evaluating Models of Memory Allocation, *ACM Transactions on Modelling of Computer Systems*, Jan. 1994.

[12] J. Bharadwaj et al., The Intel IA-64 Compiler Code Generator, *IEEE Micro*, Sept.-Oct. 2000, pp. 44-53.

[13] R. Krishnaiyer et al., An Advanced Optimizer for the IA-64 Architecture, *IEEE Micro*, Nov.-Dec. 2000.

[14] SPEC. SPEC CPU2000 Documentation (www.spec.org/osg/cpu2000/docs/)

[15] Itanium ™ Processor Benchmarks. http://www.intel.com/eBusiness/products/itanium/overview/bm012101.htm

[16] Dezso Sima. The Design Space of Register Renaming Techniques. IEEE Micro. Vol. 20 No. 5. pp. 70-83. Sep/Oct 2000

[17] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. Proc. 32nd Intl. Symp. Microarchitecture, pp. 186-192, Nov. 1999

[18] A. Gonzalez, M. Valero, J. Gonzalez and T. Monreal. Virtual Registers. Proc. Intl. Conf. High-Performance Computing, pp 364-369, 1997

[19] A. Gonzalez, J. Gonzalez and M. Valero. Virtual-Physical Registers. Proc. 4th Intl. Symp. HPCA-4, pp 175-184, Feb 1998

[20] J. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy and D. M. Tullsen. Software-directed Register Deallocation for Simultaneous Multithreaded Processors. IEEE Trans. Parallel and Dist. Systems. Vol. 10, No. 9, Sept 1999, pp 922-933.

[21] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting Dead Value Information. Proc. 30th Intl. Symp. Microarchitecture (Micro'97), Dec. 1997.

[22] D. M. Tullsen. Simulation and Modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, December 1996

[23] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, H. Wang. SoftSDV: A presilicon software development environment for the IA-64 architecture. *Intel Technology Journal*, 4th quarter, 1999.

[24] Martin C. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed Memory Machines. PhD Thesis, Princeton University Department of Computer Science, June 1996.

Efficient and Fast Data Allocation of On-chip Dual Memory Banks

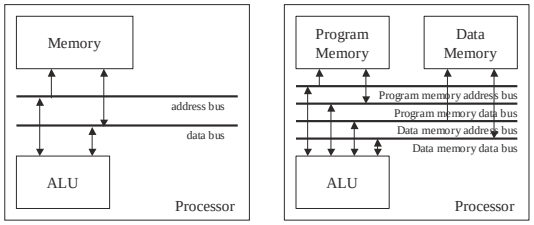
Jeonghun Cho Jinhwan Kim Yunheung Paek
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science & Technology
{jhcho,jhkim,ypaek}@soar.kaist.ac.kr

Abstract

Efficient utilization of memory space is extremely important in embedded applications. Many DSP vendors provide a dual memory bank system that allows the applications to access two memory banks simultaneously. Unfortunately, we have found that existing vendor-provided compilers cannot generate highly efficient code for dual memory space because current compiler technology is unable to fully exploit this DSP hardware feature. Thus, software developers for an embedded processor have hard time developing software by hand in assembly to exploit the hardware feature efficiently. In this paper, we present a preliminary study of a memory allocation technique for dual memory space. Through there has been some work done for dual memory banks, efficient code was generated but it required so long compilation time. Although the compilation speed is relatively of less importance for embedded processors, it still should have a reasonable upper bound particularly for industy compilers due to ever increasing demands on faster time-to-market embedded software design and implementation. To achieve such reasonable compilation speed, we simplified the dual memory bank allocation problem by decoupling our code generation into five phases: register class allocation, code compaction, memory bank assignment, register assignment and memory offset assignment. The experimental results show that our generated codes perform as good as previous work, yet reducing the compilation time dramatically.

1 Introduction

Recently, system-on-chip DSP architecture supports both on-chip and off-chip memory; the former is internal to the processor for rapid data access but its size is limited, and the latter is external to the processor for larger sized data but its speed is much slower. Since accessing the off-chip memory causes performance overhead in terms of time and energy, the code embedded in the system is generally designed hard to be fit into the on-chip memory. In our work, we focus on utilizing the on-chip memory architecture.



(a) Von Neumann Memory Architecture (b) Harvard Memory Architecture

Figure 1. The Memory Architecture

Internal memory of the DSP has Harvard memory architecture which is composed of program and data memory modules shown in Figure 1 (b). In this architecture, two memory banks are connected through two independent address and data buses. In a different way, Von Neumann memory architecture has a single memory bank with shared data and address bus shown in Figure 1 (a). One of the advantages of Harvard architecture is that it can access two memory in one instruction cycle simultaneously.

To maximize the speed of data memory access, the original design of Harvard architecture has been enhanced by many vendors of the fixed-point DSPs. In one popular design supported by Motorola DSP56000, NEC uPD77016, Analog Device ADSP2100 and DSP Group PineDSPCore, three memory banks are provided: a program memory bank plus two data memory banks each with independent address space. These three memory banks increase the memory bandwidth by allowing to access a program and two data memories in parallel. This feature of memory architecture can be shown to be very effective to many DSP algorithms such as the FIR filter algorithm ($c(n) = \sum_{i=0}^{N-1}[a(i) \times b(n-i)]$). In fact, a C implementation of the FIR filter can be executed at

an ideal rate of one tap per instruction cycle on a DSP with the three memory banks. However, this ideal speed of execution is only possible with one condition: the two variables (`a(i)` and `b(n-1)`) should be assigned to different data memory banks.

Several existing compilers that we examined were not able to fully exploit this dual memory feature, and consequently failed to generate highly optimized code for their target DSPs. This is mainly because, until recently, little work has been done by compiler researchers on data allocation techniques that efficiently utilize the dual memory architecture. This inevitably implies that the programmers should develop their applications by hand in assembly to exploit the hardware feature efficiently, which makes programming embedded DSPs quite complex and time consuming.

Probably the most recent work on this issue is done by Sudarsasnam, et al. [9]. In their work, they presented their experimental results showing that their compiler generated highly optimized code for a commercial DSP in most cases. However, the results also showed an evidence that the compilation time may increase substantially for large code or may not produce efficient code even after long exhaustive search for the optimal solution.

In this paper, we present a code generation algorithm that attempts to exploit this architectural feature more efficiently. Our algorithm is fast in that it has polynomial time complexity, and yet, as will be shown in this paper, it generates target code of as high quality as the code generated by previous work almost in all cases.

This paper is organized as follows. Section 2 describes our workbench, Motorola DSP56000 architecture. Section 3 shows our approach to support the dual memory banks. Section 4 presents our experimental results with a set of DSP benchmarks on Motorola DSP56000, and compares the performance of our compiler with Sudarsanam's, and conclusions are given in Section 5.

2 The Dual-Memory Architecture

Some existing fixed-point DSPs perform move operations in parallel for speed up operations. In this paper, as an example, we describe the Motorola DSP56000 which is one of these DSPs. The DSP56000 architectural units related to parallel move are the data ALU, the AGU, and the X/Y data memory banks.

The data ALU, shown in Figure 2, consists of four 24-bit input registers called X0, X1, Y0, and Y1, and two 56-bit accumulators called A, and B. Data transfers between the data ALU registers and the X data memory or Y data memory occur over XDB, YDB. The AGU

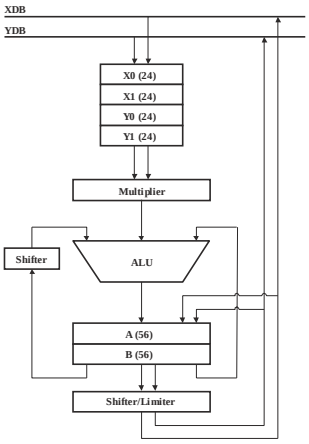


Figure 2. Data ALU of Motorola DSP56000

performs address calculations necessary to indirectly address data operands in memory. It operates simultaneously with other components to perform address calculations in parallel with the ALU operation. The AGU, shown in Figure 3, is divided into two identical halves, each of which has an address ALU and two sets of 16-bit register files. One set consists of four address registers R0 through R3 and four offset registers N0 through N3, and the other consists of four address registers R4 through R7 and four offset register N4 through N7. The two address ALUs are identical in that each contains a 16-bit full adder called an offset adder, which associated with each set can add 1) plus one, 2) minus one, 3) the contents of the respective offset register Ni, or 4) the two's complement of Ni to the contents of the selected address register Ri. The address output multiplexers select the source for the XAB, YAB. The source of each effective address may be the output of the address ALU for indexed addressing or an address register for register-indirect addressing. The X/Y data memory banks consist of two 512-word * 24-bit data memory banks, which allow at most two data memory access to occur in parallel.

Possible memory references of DSP56000 are X, Y, L, and XY. The X (Y) memory reference is that the operand, a word reference, is in X (Y) memory space. Data can be transferred from X (Y) memory to register or from a register to X (Y) memory, Long (L) memory space references both X and Y memory spaces with one operand address. XY memory space references both X and Y memory spaces with two operand addresses. Two independent addresses are used to access two word operands - one word operand is in X memory space, and one word operand is in Y memory space.

Due to the feature of the DSP56000 architecture, as mentioned above, one data ALU operation and two

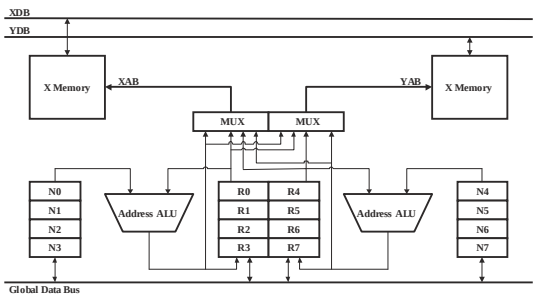


Figure 3. Address Generation Unit and X/Y Data Memory Bank

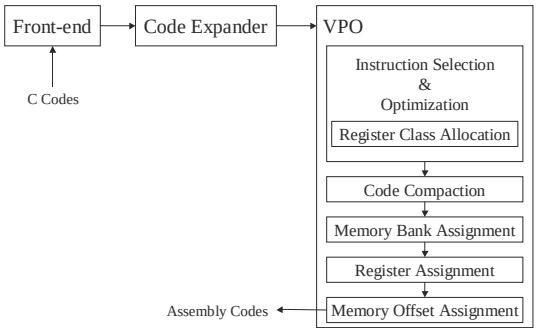


Figure 4. Overall Structure of Our Compiler

move operations may be performed in parallel during one instruction cycle. However, due to the DSP56000 microarchitecture, only the case that satisfy the following constraints can be performed: two memory access or a pair of one memory access and one register transfer are performed in parallel, destination registers are different, the X data memory access is performed with X0, X1, A, or B, and the Y data memory access is performed with Y0, Y1, A, or B.

```
int a, b, c, d, e, f;
long v, w;

v = a * b + c;
w = d * e + f;
```

Figure 5. Example C Code for Describing Our Algorithm

3 Code Generation for Dual Memory Architecture

Figure 4 shows the overall structure of our compiler. VPO is the code generator of the *Zephyr* compiler [1], which was originally developed at the University of Virginia as a part of the National Compiler Infrastructure. In this work, we have extended VPO to handle DSPs with dual memory space. In our new code generation process is divided into five phases: register class allocation, code compaction, memory bank assignment, register assignment, and memory offset assignment. As we demonstrate later, this decoupled structure of code generation phases has led us to simplify our data allocation algorithm for dual memory banks and to run the algorithm substantially faster than other previous work [9] where all these phases are coupled to handle these issues simultaneously in one phase.

Figure 5 displays an example, which will serve to illustrate the various features of the new approach in this paper. Each phase is explained in detail using this example.

3.1 Register Class Allocation

Most existing fixed-point DSPs are known to have irregular structure with *heterogeneous* register architectures. These architectures contain multiple register files where different files are usually distributed and dedicated to different sets of instructions. In our recent study [4], we showed that the graph-coloring register allocation algorithm originally implemented in Zephyr, like most existing compilers, was effective only for processors with homogeneous centeralized registers. In our work, we handle heterogenous registers by performing register allocation in two separate phases, register class allocation and register assignment (see Figure 4). In the same study, we also showed that separation of register allocation simplified our code geneartion algorithm, yet allowing us to achieving relatively good performance.

To more formally describe our register class allocation, we start this section by first presenting a few definitions.

Definition 1 *Given a target machine M, let $I = \{i_1, i_2,..., i_n\}$ be a set of all the instructions defined on M, and $R = \{r_1, r_2, ..., r_m\}$ be a set of all its registers. For instruction $i_j \in I$, we define a set of all its operands, $Op(i_j) = \{O_{j1}, O_{j2}, ..., O_{jk}\}$. Assume C_{jl} is a set of all the registers that can appear at the position of some operand $O_{jl}, 1 \leq l \leq k$. Then we say here that C_{jl} forms a* **register class** *for instruction i_j.*

For instance, SPARC has an instruction with three operands

$$\texttt{ADD}\ \ reg_i, reg_j, reg_k,$$

where all the 32 registers(r0,r1,. . .,r31) in the register

file can appear as the first operand reg_i. In this case, the set of all these registers forms a single register class for ADD. Since at the other operands reg_j and reg_k, the same 32 registers can appear, they again form the same class for the instruction. Therefore, we have only one register class defined for instruction ADD.

As another example, consider the instruction

MPY reg_i, reg_j, reg_k

of Motorola DSP56000, which multiplies the first two operands and places the product in the third operand. The DSP56000 restricts reg_i and reg_j to be input registers X0, X1, Y0, Y1, and reg_k to be accumulator A or B. In this case, we have two register classes defined for MPYA: {X0, X1, Y0, Y1} at reg_i and reg_j and {A,B} at reg_k.

Definition 2 *From Definition 1, we define S_j, a collection of distinct register classes for instruction i_j, as follows:*

$$S_j = \bigcup_{l=1}^{k} \{C_{jl}\}. \quad (1)$$

From this, we in turn define S as follows:

$$S = \bigcup_{j=1}^{n} S_j. \quad (2)$$

We say that S is the whole collection of register classes for machine M.

In the above examples, S_j for ADD and MPY are, respectively, {{r0,. . .,r31}} and {{X0, X1, Y0, Y1},{A,B}}. Notice that a typical processor with n general purpose registers like SPARC or PowerPC is often said to be homogeneous mainly because S is usually a set of a single element consisting of the n registers for the processor, which, by Definitions 1 and 2, equivalently means that the same n registers are homogeneous in all the machine instructions. In case of DSPs, however, its registers are usually dedicated differently to the machine instructions, which make them partially homogeneous only in the subsets of machine instructions. For example, notice that even one instruction like MPY of DSP56000 has two different sets of homogenous registers: XYN and AB. Table 1 shows the whole collection of register classes defined for DSP56000. In general, we say that a machine with such complex register classes has heterogeneous architecture.

The register class allocation is not to allocate real registers but to allocate a set of possible registers (that is, a register class) which can be placed as operands of an instruction. Real registers are selected among the register class for each instruction later in the register assignment phase. Since the focus of this paper is not on the register class allocation, we cannot discuss the whole algorithm here. Refer to [4] for more details.

ID	Register Class	Indicated Registers
2	XYN	X0, X1, Y0, Y1
4	XY	X, Y (long word)
5	YR	R4 ∼ R7
6	AB	Accumulator A, B
7	YN	N4 ∼ N7
8	XR	R0 ∼ R3
9	XN	N0 ∼ N3
10	X	X0, X1
11	Y	Y0, Y1

Table 1. The Register Class for Motorola DSP56000

Figure 6 shows the target code for DSP56000 translated from the code in Figure 5 after register classes are allocated. In the example, we can see the register classes associated to each register used in the code.

```
1  MOVE   a, r0
2  MOVE   b, r1
3  MOVE   c, r2
4  MAC    r0, r1, r2
5  MOVE   low(r2), low(v)
6  MOVE   high(r2), high(v)
7  MOVE   d, r3
8  MOVE   e, r4
9  MOVE   f, r5
10 MAC    r3, r4, r5
11 MOVE   low(r5), low(w)
12 MOVE   high(r5), high(w)
```

```
Register Class
r0 : XYN
r1 : XYN
r2 : AB
r3 : XYN
r4 : XYN
r5 : AB
```

Figure 6. Uncompacted Code

3.2 Code Compaction

Not only to reduce code size, but also to exploit parallel operations provided by the hardware, code compaction is performed between register class allocation and register assignment. Figure 7 shows the resulting code after code compaction is applied to the example in Figure 6.

```
MOVE          a,r0  b,r1
MOVE          c,r2  d,r3
MAC  r0, r1, r2   e,r4  f,r5
MAC  r3, r4, r5   low(r2),low(v)
MOVE          high(r2),high(v)  low(r5),low(w)
MOVE          high(r5),high(w)
```

Figure 7. Compacted Code

The compaction algorithm is based on the conventional list scheduling algorithm. The first step of the algorithm is to construct a conventional *data dependence graph* (DDG) for the current basic block. Each node of this directed graph corresponds to an instruction of uncompacted original code, and each edge represents dependence between instructions; that is, an edge e = (V_i,V_j) means that V_i must be scheduled before V_j in final compacted code.

Figure 8 shows the DDG for the original code sequence in Figure 6. The number to the left of each node represents the priority of the node that is required for scheduling sequence. The bottom node has a priority of zero, which implies that it will be scheduled last.

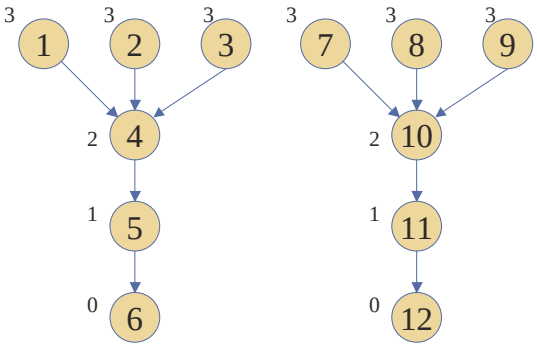


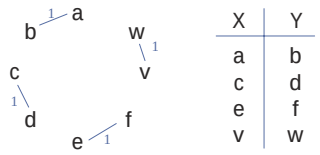
Figure 8. Data Dependence Graph for Code In Figure 6

After the DDG has been constructed, the following sequence of steps are repeatedly iterated until all DDG nodes have been scheduled:

1. Find all unscheduled DDG nodes whose parents have already been scheduled, and store them in the ready set R.
2. Sort the nodes in R in the priority order.
3. In the sorted order, fill nodes to the instruction in the compacted code until the instruction word is full[1].
4. Remove the scheduled nodes from *R*.

3.3 Memory Bank Assignment

After code compaction, each variable in the resulting code is assigned into dual memory banks (that is, X or Y in DSP56000). The first step for this, we construct a weighted undirected graph, called the *access graph* (AG). In the AG, each node corresponds to a program variable, and a pair of two nodes is connected via an edge if the corresponding two variables are scheduled into the same instruction word in the code after code compaction. Figure 9 (a) shows the AG for the code from Figure 7. The weight on an edge represents the number of a pair of the variables scheduled in a word.



(a) Access Graph (b) Assigned Memory Bank

```
MOVE             X:a,r0              Y:b,r1
MOVE             X:c,r2              Y:d,r3
MAC  r0, r1, r2  X:e,r4              Y:f,r5
MAC  r3, r4, r5  low(r2),X:low(v)
MOVE             high(r2),X:high(v)  low(r5),Y:low(w)
MOVE                                 high(r5),Y:high(w)
```

(c) Memory Bank Assignment

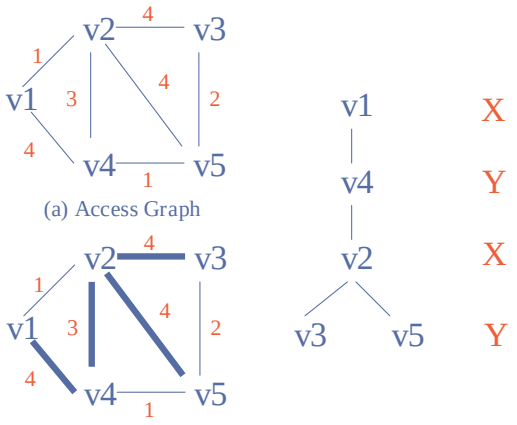
Figure 9. Result After Memory Bank Assignment

As mentioned in Section 2, if two variables referenced in an instruction word are assigned to different memories on a dual memory architecture, they can be fetched in a single instruction cycle. Otherwise, an extra cycle would be needed to fetch them in two cycles. The strategy we should take to maximize the memory throughput when we assign memory banks is, therefore, that as many as possible variables connected by an edge should be scheduled to the same instruction. Figure 9 (b) shows that the variables *a*, *c*, *e*, and *v* are assigned in X memory and the renaming ones *b*, *d*, *f*, and *w* are assigned in Y memory. This is optimal because all pairs of variables connected via edges are assigned to different memories X and Y, thus avoiding extra cycles to fetch variables, as can be seen from the code in Figure 9 (c), which is produced after memory bank assignment. In the case of variables *v* and *w*, they respectively should be moved to memory in two cycles because they are long type variables.

Unfortunately, the memory bank assignment problem that we face in reality is not always as simple as the one in Figure 9. To illustrate a more realistic and complex case of the problem, consider Figure 10, where the access graph with five variables is shown.

We view the process of assigning memory banks as the process of finding two partitions of nodes with the minimum cost. The cost here is defined to be the sum-

[1] An instruction word in DS56000 consists of one ALU operation slot and two parallel move slots. Therefore, a word can be filled with two parallel moves and one ALU operation.



(a) Access Graph

(b) Maximum Spanning Tree

Figure 10. Access Graph and Maximum Spanning Tree

mation of the total weights between the nodes in same partition when the nodes are partitioned. Based on this view, we applied a *maximum spanning tree* (MST) algorithm to solve the memory bank assignment problem. Two nodes connected by a tree edge selected by the algorithm can be simply assigned to different banks because the tree found by MST does not form a cycle. MST ensures that we can assign nodes conected via an edge with heavy weight, which implies that the bank assignment we get would be optimal.

In our compiler, Prim's algorithm [7] has been implemented to find the maximum spanning tree. The overal sequence of our memory bank assignment algorithm is shown below. This algorithm is global; that is, it is applied across basic blocks. For each node, the following sequence is repeatedly iterated until all AG nodes have been marked.

1. Select an unmarked node u in the AG. If all nodes of AG are marked, the algorithm is over.

2. Insert all edges of the selected node u in the priority queue Q and mark u.

3. If Q is empty, then go to step 1. Select the highest weighted edge $e = (u,v)$ from Q and remove e from Q.

4. If u and v are already marked, then go to step 3.

5. If either u or v is unmarked, then insert e in the spanning tree T, select the unmarked node and go to step 2.

Note in the algorithm that at least one node should always be marked in steps 4 and 5 because the edge of a marked node was inserted to Q in step 2. Figure 10 (b) shows the spanned tree obtained after this algorithm is applied to the AG given in Figure 10 (a). We can see that X memory is assigned in even depth and Y memory in odd depth in this tree.

3.4 Register Assignment

After memory banks are determined for each variable in the code, real registers are assigned to the code. Real registers are selected from those in the same register class specified in the register class allocation phase. For example, the register r0 in Figure 9 should be replaced by one real register among four candidates X0, X1, Y0 and Y1, as indicated in Table 1. However, if the instruction is a parallel move, there is an additional architectural constraint that we should consider when we assign a register: that is, data from each memory bank should be moved to a predefined set of registers. Back in the example, the variable a in the parallel move with r0 is allocated to memory X. Therefore, only registers eligible for r0 is confined to X0 and X1. If the two registers are already assigned to other instructions, register spill will occur.

Using these two types of constraints (register classes and architectural constraints), we assigned real registers to the code. Figure 11 shows the resulting code after register assignment is applied to the code shown in Figure 9 (c).

```
MOVE              X:a,X0        Y:b,Y0
MOVE              X:c,A         Y:d,Y1
MAC  X0, Y0, A    X:e,X1        Y:f,B
MAC  X1, Y1, B    A0,X:low(v)
MOVE              A1,X:high(v)  B0,Y:low(w)
MOVE                            B1,Y:high(w)
```

Figure 11. Result after Register Assignment

3.5 Memory Offset Assignment

As mentioned in Section 2, address registers are used for parallel next-address computations. To access memory locations with parallel moves, the locations should be addressed by address registers using the register-indirect addressing mode. The performance of address computations is maximized by auto-increment/decrement capabilities of AGUs because DSPs provide special hardware to efficiently support fast auto-increment/decrement addressing, thereby resulting

in higher instruction-level parallelism. However, to fully utilize this addressing, variables must be properly placed in memory.

We solved this problem, called the *simple offset problem*, using the maximum weighted path algorithm originally proposed by Leupers [5]. Figure 12 (a) shows the sequences of variable accesses in X and Y memories in Figure 11.

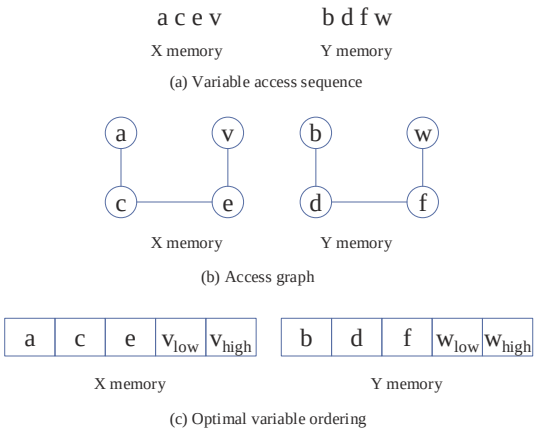


Figure 12. Memory Offset Assignment

To determine optimal variable orderings on dual memory banks, we applied the maximum weighted path algorithm to each memory bank X and Y independently. Figure 12 (b) shows the AGs constructed by access sequences, and Figure 12 (c) shows optimal variable ordering solved by maximum spanning path algorithm. The final code of our example is shown in Figure 13.

```
MOVE             X:(r1)+,X0      Y:(r5)+,Y0
MOVE             X:(r1)+,A       Y:(r5)+,Y1
MAC  X0, Y0, A   X:(r1)+,X1      Y:(r5)+,B
MAC  X1, Y1, B   A0,X:(r1)+
MOVE             A1,X:(r1)       B0,Y:(r5)+
MOVE                             B1,Y:(r5)
```

Figure 13. Result after Memory Offset Assignment

4 Comparison with Other Work

We evaluated the quality of our algorithm with a suite of DSP benchmarks on a well-known commercial DSP, the Motorola DSP56000 [6]. In this work, we take C code as the source and, as the target, produce assembly code for DSP56000 which is in turn given to the assembler to produce the machine code. We used LCC [3] as the C front-end for Zephyr.

In this section, we report performance of our algorithm with the experimental results, and analyze the effect of each major compiler technique on the performance. We also compare the best results of our current implementation with Motorola's native compiler, and identify several additional techniques.

4.1 Other work

Much work of code generation for DSPs and embedded processors has been done recently while it had not been received much attention. Because the complexity of these architectures has been increased incredibly, software developments without supporting high-level languages become so hard and impractical. In this work, Araujo and Malik [2] proposed an optimal instruction selection, register allocation, and instruction scheduling algorithm for expression trees in linear time, for DSP architecture, the Texas Instruments TMS320C25 [10]. However, this approach cannot support dual memory banks, in spite of generating efficient codes for DSPs in linear time.

Generally, the exploitation of dual memory banks was responsible for the programmer. The programmer had to allocate manually by using assembly language, and it made difficult and inefficient exploitation of dual memory banks. Code generation for dual memory banks is addressed in Saghir et al. [8]. They presented two algorithms - compaction-based (CB) data partitioning and partial data duplication. However, a DSP model featuring a large general-purpose register file is assumed.

Recently, Sudarsanam et al. [9] tried to use the dual data memory optimally using simultaneous reference allocation (memory bank + register allocation). In their work, they delayed reference allocation until after code compaction, and performed both phases of register allocation and memory bank allocation simultaneously. The constraint graph is constructed for reference allocation, and simulated annealing algorithm is applied to labelling it.

Although simultaneous optimization of register allocation and bank allocation may lead the compiler to a better solution in some cases, simulated annealing algorithm may make compilation time indefinitely too long. This long compilation time is caused by the intrinsic nature of simulated annealing that needs to search for the optimal solution from many possible candidates to be labeled in the constraint graph. Although long compilation may be tolerable in the embedded software development to some extent, compilation still needs to be done within reasonable time bounds because the longer compilation time means the slower time-to-market in software development.

4.2 Experimental Results

In this section, we present our recent experimental results that demonstrate the effectiveness of our approach by comparison with Sudarsanam et al.'s simulated annealing-based approach. For this, we selected the same code as them from DSPStone[11] and *adpcm* benchmarks:

- *complex_multiply*,
- *convolution*,
- *fir2dim*,
- *iir_biquad_N_sections*,
- *least_mean_square*,
- *matrix_multiply_1*,
- *adapt_quant*,
- *adapt_predict_1*,
- *iadpt_quant*,
- *scale_factor_1*,
- *speed_control_2*, and
- *tone_detector_1*.

The simulated annealing-based approach was experimented on a single processor in a Sun Microsystems Ultra Enterprise featuring eight UltraSPARC processors and 1016MB of RAM. On the other hand, our approach was experimented on a Intel Pentium III 666MHz and 512MB of RAM. The experiments were conducted on different machine platforms because the compiler development environments for both approaches were different.

Table 2 compares the quality of code generated by both approaches. The performance figures of simultaneous reference allocation in Table 2 are from their literature [9]. The results showed that we could achieve similar quality compared with previous work, while our compiler was incredibly fast. In Table 2, we could see that the average improvement in code size and compilation time due to our approach was 14.6% and 0.05 sec, respectively. From these benchmark results, we found when quality of uncompacted code was relatively low, the improvement in the code size was high. For instance, $complex\_multiply$ and $least\_mean\_square$ showed high percentages of improvement, but on the other hand, quality of uncompacted code showed poor results. This caused that redundant code gave many chances to compact the code in code compaction phase.

In simultaneous reference allocation, many different types of costs and complexities represented in their constraint graph should be combined and solved simultaneously. This inevitably increased the complexity of the problem of finding an optimal solution to memory bank and register allocation. This increased complexity of the problem led their approach to resort to simulated annealing which explores the search space indefinitely until it finds a reasonably optimal solution. According to our analysis, the time complexity of our approach using Prim's algorithm [7] for maximum spanning tree is $O(nlogn)$. This is partially due to our decoupled structure of code generation phases. As discussed earlier, by decoupling the code generation into several phases, we were able to simplify the complexity of the problem, yet still achieving reasonable optimal solutions.

5 Conclusions & Future Work

Many DSP vendors provide a dual memory bank system which allows the applications to access two memory banks simultaneously. Unfortunately, several existing compilers were not able to fully exploit this dual memory feature. In this paper, we proposed decoupled approach for supporting dual memory architecture: *register class allocation*, *code compaction*, *memory bank assignment*, *register assignment*, and *memory offset assignment*. This decoupled structure of code generation phases led us to simplify our data allocation algorithm for dual memory banks and to run the algorithm in reasonable time. The experimental results showed that we achieved the comparable results in code size and the enhanced results considerably in compilation time to related work.

A number of interesting topics still remain open for future work. For instance, generally, an AGU has address registers, mode registers, and offset registers. To exploit all these registers efficiently, a suitable algorithm for address register allocation is required, and an interprocedural analysis for passing arguments is indispensable in calling convention of dual memory architecture because the caller have to know memory access pattern of callee for passing arguments.

References

[1] A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Technical Report at `http://www.cs.virgina.edu/zephyr`, University of Virginia, 1998.

[2] G. Araujo and S. Malik. Code Generation for Fixed-point DSPs. *ACM Transactions on Design*

Benchmark	Simultaneous reference allocation				Our approach			
	Size (uncomacted code)	Size (comacted code)	Code Size Imprv.	Time (sec)	Size (uncompacted code)	Size (compacted code)	Code Size Imprv.	Time (sec)
complex_multiply	33	**30**	9.1%	2	55	**40**	27.2%	0.04
convolution	49	**45**	8.2%	308	40	**33**	17.5%	0.02
fir2dim	178	**158**	11.2%	5482	109	**96**	11.9%	0.07
iir_biquad_N_sections	132	**128**	3.0%	1632	124	**102**	17.7%	0.03
least_mean_square	115	**102**	11.3%	2776	176	**137**	22.1%	0.06
matrix_multiply_1	89	**85**	4.5%	1011	129	**105**	18.6%	0.07
adapt_quant	235	**227**	3.4%	4399	206	**197**	4.3%	0.03
adapt_predict_1	231	**209**	9.5%	8105	247	**216**	12.5%	0.07
iadpt_quant	85	**81**	4.7%	324	63	**54**	14.2%	0.05
scale_factor_1	74	**66**	10.8%	266	60	**55**	8.3%	0.04
speed_control_2	277	**247**	10.8%	5217	217	**199**	8.2%	0.1
tone_detector_1	84	**75**	10.7%	536	75	**65**	13.3%	0.06

Table 2. Results for DSPStone Benchmark Programs

Automation of Electronic Systems, 3(2):136–161, April 1998.

[3] C. Fraser. A Retargetable Compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[4] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84–92, Nov. 2001.

[5] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Internaltional Conference on Computer-Aided Design*, 1996.

[6] Motorola Inc., Austin, TX. *DSP56000 24-Bit Digital signal Processor Family Manual*, 1995.

[7] R. Prim. Shortest Connection Networks and Some Generalizations. *Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[8] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGOPS Operating Systems*, pages 234–243, 1996.

[9] A. Sudarsanam and S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):242–264, April 2000.

[10] Texas Instruments, Austin, TX. *TMS320C2x User's Guide. Revision C*, 1993.

[11] V. Zivoljnovic, J.M. Velarde, C. Schager, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.

Code Size Efficiency in Global Scheduling for ILP Processors

Huiyang Zhou Thomas M. Conte
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
North Carolina State University
{hzhou,conte}@eos.ncsu.edu

Abstract

In global scheduling for ILP processors, region-enlarging optimizations, especially tail duplication, are commonly used. The code size increase due to such optimizations, however, raises serious concerns about the affected I-cache and TLB performance. In this paper, we propose a quantitative measure of the code size efficiency at compile time for any code size related optimization. Then, based on the efficiency of tail duplication, we propose the solutions to two related problems: (1) how to achieve the best performance for a given code size increase, (2) how to get the optimal code size efficiency for any program. Our study shows that code size increase has a significant but varying impact on IPC, e.g., the first 2% code size increase results in 18.5% increase in static IPC, but less than 1% when the given code size further increases from 20% to 30%. We then use this feature to define the optimal code size efficiency and to derive a simple, yet robust threshold scheme finding it. The experimental results using SPECint95 benchmarks show that this threshold scheme finds the optimal efficiency accurately. While the optimal efficiency results show an average increase of 2% in code size, the improved I-cache performance is observed and a speedup of 17% over the natural treegion results is achieved.

1. Introduction

The I-cache performance for an application is determined by its working set size. If the program size is exceedingly large compared to the I-cache or TLB size, it may result in high miss rates, which in turn degrades the performance of the processor. On the other hand, in the scheduling phase of an ILP (instruction-level-parallelism) compiler, there is a lot of effort placed on enhancing the performance by exploiting the available ILP. As larger scheduling regions tend to provide more ILP, region-enlarging optimizations are commonly used in or before the instruction scheduler. However, those optimizations often cause an increase in static code size. Loop unrolling and loop peeling are examples of such optimizations in cyclic scheduling. In acyclic global scheduling, tail duplication (or code replication) is the most commonly used region enlarging / ILP enhancing optimization. Even with its evident impact on code size increase, it is applied due to its capability to remove the side entries of a trace [5], [13] and to avoid the conditional / unconditional branches [12]. Our experience is that other code size related optimizations in acyclic scheduling, such as code downward motion through branches and recovery code for speculations [15], have less impacts on both ILP and code size than tail duplication.

In the paper, we study the *code size efficiency* of code-size-related optimizations in acyclic scheduling, especially the tail duplication. We then present a very efficient way of regulating tail duplication for global instruction scheduling. To do this, we first define a quantitative measure of the code size efficiency that is for *any* code size related optimization. The measure is calculated as the ratio of ILP improvement (in terms of static IPC) to code size increase. The static IPC is the instruction-per-cycle measured at compile time to show the ILP exploitation based on instruction scheduling. Based on this general description, two more specific definitions are formulated: *average code size efficiency* and *instantaneous code size efficiency*. The average code size efficiency measures the ILP improvement at the cost of code size for overall applications of code size related optimizations. The instantaneous code size efficiency is used for an individual application of an optimization based on the current code size.

As the static IPC is hard to calculate before the schedule time, we propose a heuristic to estimate the *expected execution time* of a multi-path region using a *dependence bound* and a *resource bound*. The experimental results show that the treegion scheduler

produces schedules very close to the expected execution time (92% to 97% accuracy). Then, two related problems are investigated based on the instantaneous code size efficiency of different tail duplication candidates: (1) how to achieve *the best speedup for a given size code increase*, i.e., how to get the best average code size efficiency for a given code size; and, (2) how to *get the optimal code size efficiency for any program*. To find the solution to the first problem, all the possible tail duplication candidates in the program scope are ordered based on their instantaneous code size efficiency. The candidates are then chosen based on this order until the estimated code size limit is reached. The simulation results using SPECint95 show that for a modest pre-scheduling code size increase of 2% over the original size, the scheduled code gains 18.5% speedup and a 1.6% code size *decrease*[1]. Another observation from the simulation results is that for any benchmark, the initial code size increase over the original has a much larger impact on static IPC than the same increase over an already bloated program— e.g., the initial 2% code size increase result in IPC change of 18.5%, while the IPC change is less than 1% when pre-scheduling code size limit varying from 20% to 30%.

Based on above observations, we define the *optimal code size efficiency for a program* and propose a simple, yet robust threshold scheme to find the optimal solution. This threshold is derived mathematically to be the code size efficiency measure that we proposed before. The robustness of the scheme (i.e., the effective range of the threshold) is determined by the rate of static IPC change over code-size increase around the optimal solution. The simulation results show that this simple threshold scheme finds the optimal solution for every benchmark with average post-scheduling 2% code size increase over the original size. When taking the cache effects and branch prediction impact into account, it results in a 4% decrease on I-cache miss penalties (for a 32KB I-cache), due to the increased sequential locality and more compact schedule, and a 17% speedup overall over the natural treegion results (treegion without any tail duplication). The experiment with different I-cache sizes shows that the speedup also holds for both small I-caches of 16KB and large I-caches of 64KB [21].

The remainder of the paper is organized as follows. Section 2 briefly introduces the treegion-based global scheduling, and the simulation methodology of the experiments. The quantitative measures of the code size efficiency are discussed in Section 3. The optimal tail duplication for scheduling under a given code size constraint is contained in Section 4.1 and the solution to the optimal code size efficiency is discussed in Section 4.2. Finally, Section 5 concludes the paper.

[1] This decrease is due to the general operation combining [4] exploited by our global scheduler.

2. Treegion-based global scheduling and simulation methodology

2.1. Treegions and treegion-based global scheduling

In this paper, treegion-based global scheduling [1],[2] is used as the acyclic scheduling framework. However, it needs to be pointed out that although the experimental results are obtained using treegion scheduling, the same methodology of this code size efficiency study is applicable to other global scheduling approaches, such as superblock scheduling [5] and hyperblock scheduling [7].

Treegion-based global scheduling aims for high performance for wide issue VLIW / EPIC processors although it can be applied to superscalar processors as well. It has two steps: treegion formation [1] and tree traversal scheduling (TTS) [2]. A treegion is a single-entry / multiple-exit nonlinear region that consists of basic blocks (BBs) with control-flow forming a tree, as illustrated in Figure 1a. Based on the control flow graph (CFG) in the Figure, two treegions are formed. The treegions that are formed without any tail duplication are referred to as *natural treegions*. When the tail duplication is applied, a larger treegion can be formed. For the example CFG in Figure 1a, after the BB7, BB8, and BB9 are duplicated and the corresponding unconditional branches are removed, one treegion is formed containing all the BBs in the CFG, as shown in Figure 1b. The trade-off for exposing ILP through treegion formation is the code-expansion that results from duplicates of BB7, BB8 and BB9. Note that in this paper, the tail duplication is performed on the unit of natural treegion (i.e., merge points), e.g., in the example of Figure 1, the entire treegion 2 is duplicated instead of the BB7. In the previous treegion scheduling works, the tail duplication is performed based on a heuristic discussed in [1], which we refer to as Havanki's heuristic and briefly describe it as follows. Havanki's tail duplication heuristic is based on several factors: code expansion limit, path count (the number of paths in a treegion) and the number of the incoming edges to a merge point. The code expansion limit is a global control parameter, while the other two are based on the topology of the CFG. When any of those limits is reached, the tail duplication will stop and a new treegion will be formed. The advantage of this heuristic is that it solely depends on the topology of the CFG and it is not susceptible with the profiling errors.

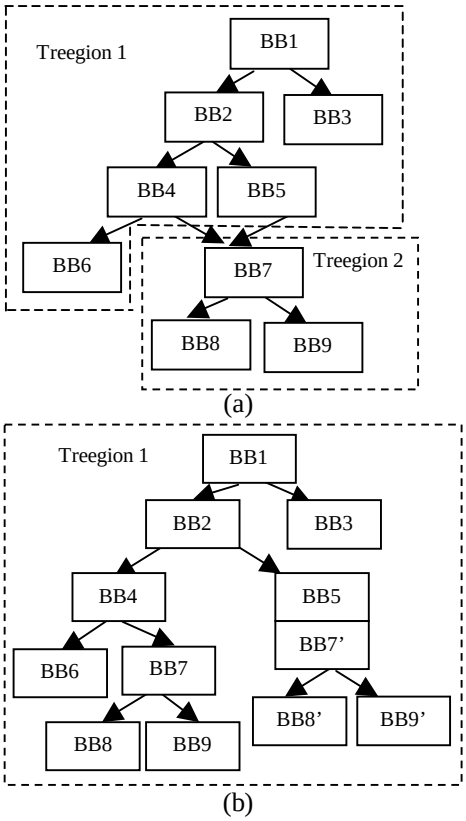


Figure 1. (a) The CFG and the treegions constructed; (b) The treegion constructed after the tail duplication

During the tree traversal scheduling (TTS), the BBs are scheduled in a predetermined traversal order based on treegion topology and profile information. When a BB is currently being scheduled, those instructions that are dominated by the BB will be considered as scheduling candidates until the block-ending branch is scheduled. Those candidate operations are scheduled based on an order determined by a heuristic that includes their execution frequency, exit count, and data dependence height. The details of tree traversal scheduling can be found in [2],[4].

2.2. The code size increase in treegion scheduling

In treegion based scheduling, most code size increase is from tail duplication during treegion formation[2]. In TTS, downward code motion and general operation combining also contribute to code size changes. Downward code motion happens when the block-ending branch is scheduled earlier than some instructions in the same BB. To maintain the semantics of the program, those instructions need to be placed at every possible exit path of the branch, which may introduce some code replication. In TTS, this downward code motion is combined with partial dead code removal so that only instructions producing a variable live at both exit paths will be replicated. The general operation combining is used at scheduling time to remove redundant operations. When one operation is selected for scheduling, it is compared to other operations that have already been scheduled in the same cycle. If a scheduled operation is found to have the same opcode and source operands, the candidate operation is then merged into the scheduled operation with necessary renaming. Since a treegion contains multiple execution paths, it exploits more opportunities for general operation combining than those of linear regions. As a result, the scheduled code will have a reduced code size. When both downward code motion and general operation combining are used, the benchmarks in SPECint95 show an average of 3.5% code size decrease for treegions formed without any tail duplication (i.e., using natural treegions). When tail duplication is performed, there are more chances for general operation combining. For the treegions formed using Havanki's heuristic, 12.8% code size decrease is observed at scheduling time while the effective overall code size increase is about 70% (i.e., the code size increase would be 82.8% without general operation combining).

2.3. Simulation methodology

The algorithms for the code size efficiency study in this paper and for treegion based global scheduling are implemented in LEGO compiler [11], a research ILP compiler developed for high performance VLIW / EPIC [9] style microprocessors at North Carolina State University. The compiling process of LEGO compiler is as follows. All programs are first compiled with classic optimizations using either (1) the IMPACT compiler from University of Illinois [10] and converted to Rebel textual intermediate representation using the Elcor compiler from Hewlett-Packard Laboratories [8], or (2) read directly from IA-64 assembly generated from the Intel or GCC compilers. Then, the LEGO compiler is used to profile code, form treegions and schedule the instructions. After instrumentation is added for trace-based timing simulation, the scheduled intermediate code is either converted into an inline execution simulator that is emitted as C code (the technique used in this paper) or emitted as IA-64 assembly. Finally, a trace-based timing simulation runs together with an execution simulation to obtain the simulation results while ensuring the correctness of the program. In our experiments, all benchmarks in SPEC95int suite run to completion.

[2] A small additional code size increase is caused by copy operations to preserve liveness beyond the treegion scope.

For the simplicity, an 8-way universal issue machine model is used in this study. The specification of the model is show in Table 1.

Table 1. The specification of the machine model used in the experiment

	Specification
Execution	Dispatch/Issue/Retire bandwidth: 8; Universal function units: 8; Operation latency: ALU, ST, BR: 1 cycle; LD, floating-point (FP) add/subtract: 2 cycles.
I-cache	Compressed (zero-nop) and two banks with 2-way 16KB each bank [19]. Line size: 16 operations with 4 bytes each operation. Miss latency: 12 cycles
D-cache	Size/Associativity/Replacement: 64KB/4-way/LRU Line size: 32 bytes Miss Penalty: 14 cycles
Branch Predictor	G-share style Multiway branch prediction [20] Branch prediction table: 2^{14} entries; Branch target buffer: 2^{14} entries/8-way/LRU. Branch misprediction penalty: 10 cycles

3. The quantitative measure of code size efficiency

3.1. Code size efficiency for code size related optimizations in global scheduling

The motivation of a region enlarging optimization in global scheduling is based on the premise that larger scheduling regions can exploit more ILP. With tail duplication as an example optimization, Figure 2 shows the relationship between static code size and performance for the benchmark *compress*. Note that although the working size of *compress* is small, it exemplifies the relationship between the code size and ILP exploitation that are shared by other larger benchmarks. The experimental results in Figure 2 show code sizes vs. ILP for BB scheduling and treegion scheduling. For treegion scheduling, three possible tail duplication strategies are presented: natural treegions, tail duplication based on Havanki's heuristics, and tail duplication for all the possible merge points that have execution frequency larger than zero ('All_Possible'). In the experiment, the ILP is measured using *static IPC*, which is the instruction-per-cycle estimated at compile time to show the ILP exploitation based on instruction scheduling. Also, when calculating this static IPC, the dynamic instruction count (IC) based on BB scheduling code is used for treegion-scheduling results to show the effective IPC. The code size is measured using the relative ratio, i.e., the ratio of resulted code size over the original code size.

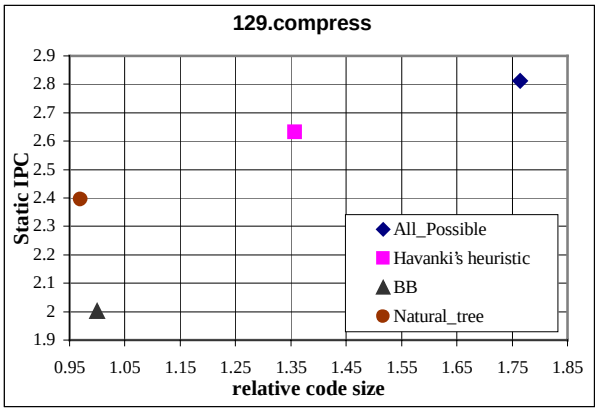


Figure 2. The relationship between performance and static code size for benchmark *compress*

As shown in Figure 2, natural treegion scheduling shows a 3% code size decrease over the original code size (the decrease is due to the general operation combining of TTS) and 20% speedup over BB scheduling. If tail duplication is applied, more ILP will be exploited (up to another 21% speedup) in the global scheduling phase with the cost of an increase in code size (up to 76%). Base on these observations, it seems that the natural treegion is a good starting point for code size related optimization, and that the ratio of the change in static IPC over the change in code size provide a reasonable measure of the efficiency of the code size expanding optimizations at compile time. It is noted here that although the dynamic IPC is more representative of the real performance, it depends on many factors including the branch prediction accuracy, cache performance, code layout and other optimizations, which are hard to quantify at compile time. The static IPC, on the other hand, indicates the ILP exploitation at compile time and is the goal to maximize with compile-time optimizations. So, the static IPC is used as the performance indicator in our measure of the tradeoff between ILP exploitation and the code size increase and the dynamic IPC effects are examined in Section 4.2.

Here, we define two different types of code size efficiency based on different forms of the ratio of IPC changes over relative code size changes.

Average code size efficiency: This type of efficiency provides a measure of the average ILP provided by code size related optimizations at the cost of a unit code size increase and it is defined as follows:

$$Efficiency_{ave} = \frac{IPC_{candidate} - IPC_{natural\_treegion}}{code\_size_{candidate} - code\_size_{natural\_treegion}} \quad \textbf{(1)}$$

In Equation (1), the term $(IPC_{candidate} - IPC_{natural\_treegion})$ represents the ILP

improvement of the candidate optimizations and the term ($code\_size_{candidate} - code\_size_{natural\_treegion}$) represents the cost of such optimizations in terms of static code size. Graphically in Figure 2, the average code size efficiency represents the slope of a line connecting the natural treegion result and the one under consideration (i.e., 'candidate'). With this quantitative measure, the comparison can be made for different code size related optimizations and for the different applications of the same optimization. For example, based on tail duplication results in Figure 2, it can be seen that the Havanki's heuristic produces a slightly better code size efficiency than duplicating all the possible candidates. Note that if the efficiency of an optimization is calculated as negative, it represents one of two extreme cases: (a) the optimization increases the IPC and decreases the code size— this optimization should always be applied, or (b) the optimization decreases the IPC at the cost of more code size— this optimization usually needs to be avoided.

Instantaneous code size efficiency: this type of efficiency measures the ILP improvement of an *individual* application of an optimization based on the current code size, and it is defined as follows:

$$Efficiency_{inst} = \frac{IPC_{after\_indivial\_application} - IPC_{before\_indivial\_application}}{code\_size_{after\_indivial\_application} - code\_size_{before\_indivial\_application}} \quad \textbf{(2)}$$

Using the tail duplication as an example optimization, there could be many merge points in a program as candidates for this optimization. Then, for each possible tail duplication (i.e., an individual application), there is an instantaneous efficiency associated with it.

For the tail duplication example in Figure 2, if we imagine that there is a curve representing the relationship between IPC and code size of tail duplication optimization, the instantaneous efficiency is the tangent slope of the curve (i.e., the derivative of the curve) at the point corresponding to the current code size. The average code size efficiency can then be viewed as the effect of averaging the instantaneous efficiency of all the tail duplications that occurred in global scheduling.

3.2. A heuristic to compute efficiency using expected execution time

Since the code size efficiency calculation requires the (static) IPC measurement, which is not known before the schedule time, we propose a heuristic to compute the expected execution time so that the IPC changes can be approximated by the changes in expected execution time. This heuristic is based on the *data dependence bound* and *resource bound* and is defined as Equation 3 for a multi-path region, e.g., a treegion.

$$Exe\_Time_{Expected} = \sum_{path\_i} \left[Max\left(data\_dependence\_bound_{path\_i}, resource\_bound_{path\_i} \right) * Freq_{path\_i} \right] \quad \textbf{(3)}$$

In Equation 3, the expected execution time of a region is computed as the sum of the expected execution time of each path, which is in turn computed as maximum of the data dependence bound and the resource bound of the path. Similar to the performance bounds proposed in [14], [17], we use the true data dependence height of Data Dependence Graph (DDG) as the dependence bound. The resource bound is calculated using a technique similar to the ResMII calculation from iterative modulo scheduling [16]. The execution frequency for each path, $Freq_{path\_i}$, is obtained from profiling information.

The effectiveness of this heuristic is verified by comparing the expected execution time to the treegion scheduled results, as shown in Table 2. Here, the execution time of the scheduled code is measured using a scoreboard-based simulation, which enforces the data dependence and resource dependence. In the benchmark *gcc*, for example, the overall execution time based on scheduled result is 7.5% larger than the expected execution time using this heuristic. The mismatch is because the data dependence bound is calculated assuming all the false register dependencies can be removed by software renaming, and that the control dependencies can be minimized by treegion multiway branch transformations [4]. This assumption is too optimistic as liveness beyond the BB scope may require a copy instruction to be inserted. Also, the renaming may not be applicable to some special purpose registers, such as parameter passing registers.

3.3. The code size efficiency for tail duplication optimization

When we consider tail duplication as the optimization of interest, for each control edge entering a merge point, we can calculate its instantaneous code size efficiency

Table 2. The accuracy of the heuristic to compute the expected execution time

Benchmark	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
Ratio of execution time based on scheduled code over expected execution time	1.036	1.075	1.078	1.047	1.071	1.067	1.081	1.063

using Equation 2 so that we can selectively apply the tail duplications based on their efficiencies. In treegion-formed code, four types of tail duplication candidate can be encountered based on the dominance relationship and number of edges entering the merge point, as shown in Figure 3.

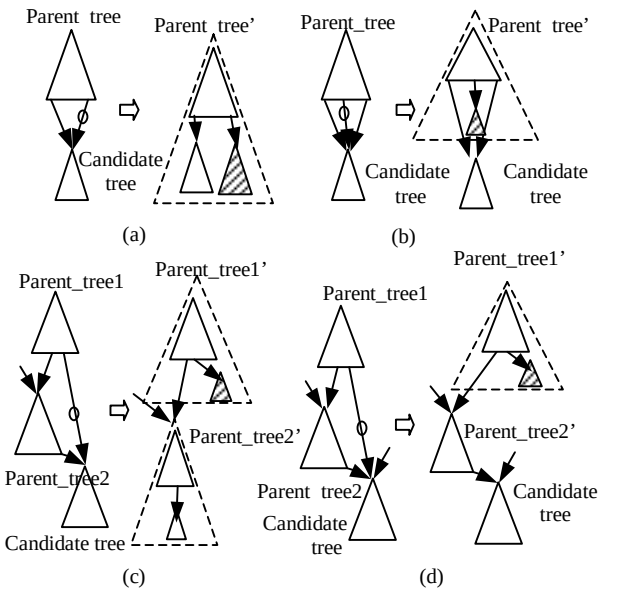


Figure 3. Four types of possible tail duplication in treegions (the edge marked with 'o' representing the edge to be removed by the candidate tail duplication, the shaded treegion represents the duplicated region): (a) Type-1: The parent tree dominates the candidate tree and there are 2 edges entering the candidate tree; (b) Type-2: The parent tree dominates the candidate tree and there are more than 2 edges entering the candidate tree; (c) Type-3: The parent tree does not dominate the candidate tree and there are 2 edges entering the candidate tree; and (d) Type-4: The parent tree does not dominate the candidate tree and there are more than 2 edges entering the candidate tree.

As shown in Figure 3, after the type-1 tail duplication, the resulted treegion (the parent_tree' in the dashed line) will absorb both the original and the duplicate copy of the candidate tree. For type-3 tail duplication, the original candidate tree will be absorbed into parent tree 2 and the duplicate will be included in the parent tree 1. For the other two types, only the duplicate of the candidate tree will be absorbed.

4. Optimal code size efficiency in global scheduling

Based on the quantitative measures of the code size efficiency of code size related optimizations such as tail duplication, one useful goal is to find the optimal code size efficiency achievable for the optimization. The term '*optimal*' here has two different meanings: (a) if there exists a limit on code size, the optimal solution is maximizing the IPC while satisfying the code size constraint (i.e., find the best average code size efficiency for a given code size). Although code size constraints are more common in embedded processors [18] than high performance EPIC processors, it is useful when we want to limit the whole or working program size (i.e., the part of the code with execution frequency larger than zero) below the level-1 I-cache size. The solution to it can be represented using a curve showing the best possible IPC for any code size. The second 'optimal' meaning is (b) if there is no such a code size limit, the optimal solution is a good trade-off between ILP and code size so that the IPC is maximized at the minimal cost of code size increase. The meaning of this best trade-off will be clear once we obtain the curve of best IPC vs. code size based the solution to (a). Using the tail duplication as an example code-size-related optimization, Section 4.1 provides an algorithm to find the best efficiency for a given code size, and Section 4.2 defines the optimal efficiency problem without code size constraints and derives a simple, yet robust threshold scheme.

4.1. Optimal code size efficiency for a given code size limit

In order to find best code size efficiency of a given code size for global scheduling using tail duplication, we first compute the instantaneous code size efficiency for all possible tail duplication candidates. Then, the candidates are selected based on their efficiencies until the size constraint is reached. The detailed algorithm is shown in Figure 4. As shown in Figure 4, we use an iterative approach for tail duplication. In each iteration of steps 2 and 3, the candidate with best instantaneous code size efficiency will be chosen and performed if such a tail duplication will not exceed the code size constraint. Although it may be possible to find the 'real' optimal solution (i.e., tail duplications with best IPC) with an exhaustive search algorithm, like what used in determining best function inlining under a code size limit [18], the complexity of such a search approach is further increased by the fact that one tail-duplication may change the efficiency of other candidates and increase the number of the possible tail duplications.

Table 3. The base code size and IPC for each benchmark

Benchmark	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
Static Operation Count	1439	368960	59853	40835	14487	33629	76026	149751
Static IPC	2.395	2.24	1.86	2.49	2.0	2.03	2.19	2.51

The algorithm described in Figure 4 was implemented in LEGO compiler and experimented on SPECint95 benchmarks. Table 3 shows the base static IPC (using natural treegion scheduling) and the original static code size in terms of operation count for each benchmark. Figure 5 shows the experimental results of benchmark *compress* where the target code size increases are 0% (i.e., natural treegion), 2%, 5%, 10%, 15%, 20%, 30%, and 80%. The results for tail duplication based on Havanki's heuristics are also included. Note that due to the effect of the general operation combining in TTS, the scheduled code size is actually less than the target size.

Algorithm for optimal tail duplications under code size constraints

0. Mark the loop edges so that the tail duplication will not overlap with cyclic optimization such as loop unrolling.
1. Calculate the instantaneous code size efficiency for all possible tail duplication candidates in the program scope.
2. Find the one with best code size efficiency.
3. If the selected candidate satisfies the code size constraint, perform the tail duplication and update the code size efficiencies of the candidates that are affected by the tail duplication process.
4. Repeat steps 2-3 until the code size limit is reached.

Figure 4. The algorithm for best tail duplication for global scheduling under code size constraints

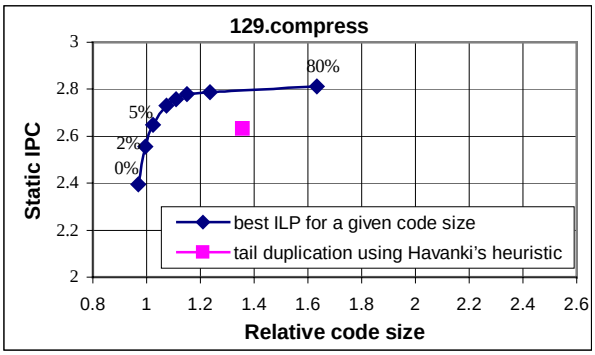


Figure 5. The relationship of ILP vs. code size of benchmark *compress*

Several important observations can be made from Figure 5. First, the code size increase due to tail duplication has significant impact on ILP, e.g., performing tail duplication up to 5% of its original size will results in 10.6% speedup and 2.4% increase in scheduled code size over the original code size. Comparing to the tail duplication based on Havanki's heuristics in traditional treegion formation, the code size efficiency is greatly improved by the increased IPC and decreased code size. There are two main reasons for relatively low efficiency of Havanki's heuristic. First, the heuristic is mainly based on local features and does not account for the profile information. When the treegion formation starts, the treegion expands by tail duplication until the path count limit / code size limit is reached or there are too many incoming edges at the next merge point. As a result, it duplicates many codes that have low execution frequency and fail to do so for some basic blocks or small treegions with high execution frequency. For example, in Figure 3b, if the number of the incoming edges to the candidate tree is beyond the predetermined limit, the candidate tree will not be duplicated even it has a high execution frequency. Secondly, Harvanki's heuristic does not take account of the potential speedup when making a decision of whether a candidate should be duplicated. As a result, it may choose to duplicate and combine treegions that do not have reduced schedule length.

Another important observation based on Figure 5 is that the impact on ILP of code size *decreases* rapidly as given code size *increases*, e.g., the first 2% code size results in 7% IPC changes, while code size increase from 20% to 30% only results less than 0.5% IPC changes. This phenomenon is expected because it is a known fact that most (e.g., 90%) of the execution time is spent on a small amount (e.g., 10%) of the static code for many programs. As a result, once we finish duplicating tail treegions in that small amount (10%) of the code, further duplications will have relatively small effects on execution time, (i.e., those tail duplications will have low instantaneous code size efficiencies). This feature are also verified with other benchmarks in our experiments, e.g., the relation between ILP and code size of the benchmark *vortex* (the notorious benchmark *gcc* has a very similar curve), as shown in Figure 6, where the target code size increases are 0%, 2%, 5%, 10%, 20 %, 30%, and 80%.

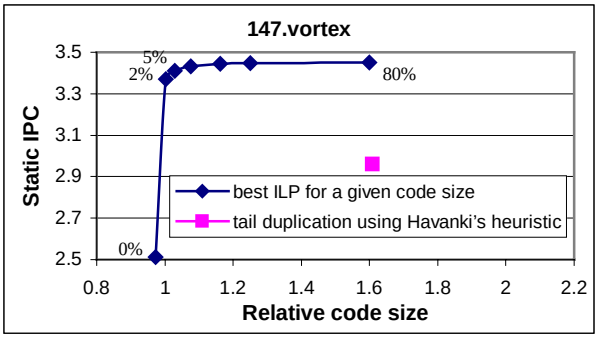


Figure 6. The relationship of ILP vs. code size of benchmark *vortex*

Figure 6 shows the dramatic IPC change (around 34%) for the first 2% code size increase, which also shows 14% speedup and 60% less code size over the traditional treegion formation approach. Two interesting observations can be made from Figure 5 and 6. First, the initial code size increase show much more IPC improvements in benchmark *vortex* than in benchmark *compress*, which means the tail duplications resulting in the initial code size increase in *vortex* have much higher efficiency than those in *compress*. The high efficiency of those tail duplications in *vortex*, based on our analysis of the program, is mainly due to high execution frequency of those codes (i.e., in the heavily executed portion of *vortex*, many control edges are worthwhile to be removed by tail duplication). Secondly, the 'diminishing returns' happen quickly for benchmark *vortex*, after the code size increase beyond 2%, comparing to benchmark *compress*, which suggests that for benchmark *vortex* a smaller percentage of code is frequently executed than benchmark *compress*. This can be verified with the statistical characteristics of the program, as shown in Table 4. From Table 4, it can be seen that higher percentage of the code of benchmark *vortex* are infrequently executed than benchmark *compress*. Given 2% code size increase for *vortex*, the portion of the program with high execution frequency has been explored for possible tail duplications while for *compress*, such code size increase is just not enough for the possible candidates in frequently executed portions.

In terms of the average of all benchmarks, the initial 2% code size increase results in 18.5% speedup over natural treegion and 1.6% code size decrease over the original code size.

4.2. Finding the best code size efficiency for global scheduling using tail duplication

Based on the characteristics of the curve representing the relationship between best IPC and code size, especially the 'diminishing returns' phenomenon, we can define the '*best code size efficiency*' as the point where the diminishing returns starts, as point *A* (i.e., the knee of the curve) shown in the exemplary ILP vs. code size curve in Figure 7.

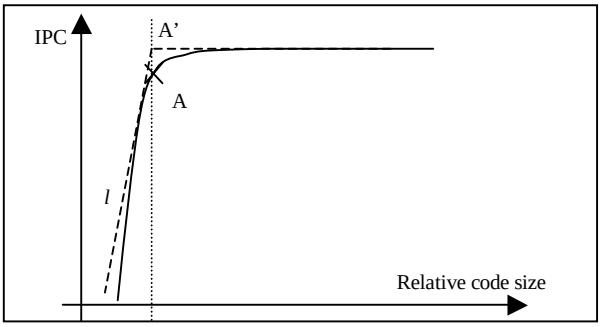


Figure 7. The solution to optimal code size efficiency

In consideration of how to find this optimal point along the curve, we can first simplify the curve as two straight lines (as the two dashed lines in Figure 7) and the optimal solution then becomes point *A*'. In order to find *A*', we can use a threshold on the first derivative of the curve, which will have a shape of bold solid lines shown in Figure 8.

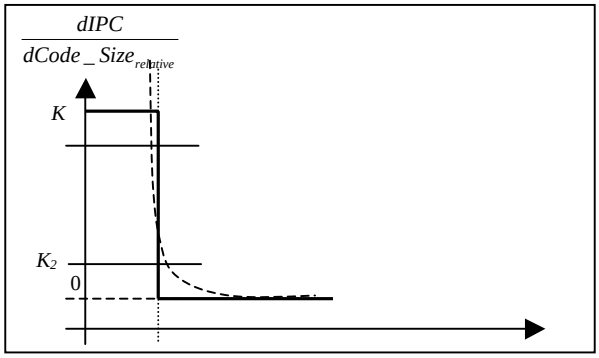


Figure 8. The derivative of the IPC vs. code size curve

From Figure 8, it can be seen that point A' can be found with a threshold on the first derivative of the IPC

Table 4. The statistics of operations with different execution frequencies

Benchmark	Maximal Execution Frequency (MEF)	Percentage of ops with execution frequency < 0.01%*MEF	Percentage of ops with execution frequency < 0.1%*MEF	Percentage of ops with execution frequency < 1%*MEF
compress	0.4 Million	55.04%	64.07%	64.26%
vortex	12 Million	84.32%	92.37%	98.45%

over code size and the threshold can be anywhere between zero and *K*, where *K* is the slope of the line *l* in Figure 7. In other words, the slope K determines the *robustness* of the threshold scheme. Since the real IPC vs. code size curve is not linear, its derivative will take a shape similar to the curve in dashed lines in Figure 8. Although the effective threshold range (i.e., the robustness) is decreased, say to from K2 to K1, it is still a relative large range due to the large slope of the IPC vs. code size curve around the 'knee' point. Thus, a large variation in the threshold on the first derivative from K1 to K2 will only result in relatively small variations from optimal point A.

As mentioned in Section 3.1, the instantaneous code size efficiency is actually the first derivative of the IPC vs. code size curve. So, this scheme becomes simply a threshold on the instantaneous code size efficiency and this threshold can be any value between K1 and K2. The meaning of K1 and K2 can be described in Figure 9, which is the zoomed area around the optimal point A in Figure 7. In Figure 9, points B and C are close to optimal solution, point A, and they represents the region of acceptable solutions. Then, the instantaneous code size efficiencies of point B and C (i.e., the slopes of the dashed lines l_1 and l_2 in Figure 9) determines the robustness of the threshold scheme.

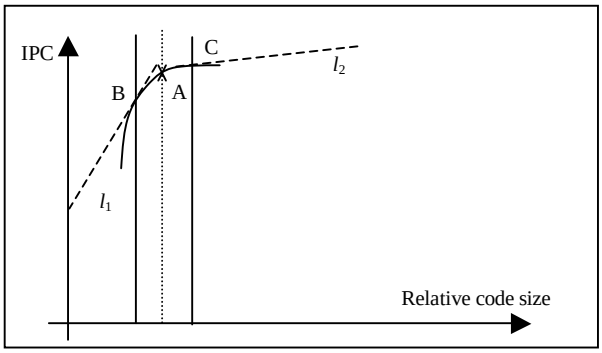


Figure 9. The robustness of the threshold scheme (determined by the slope of the tangent lines at points B and C)

As the expected execution time is used to approximate the static IPC, the threshold scheme on instantaneous code size efficiency can be further derived as a threshold on the ratio of changes in execution time over changes in code size (the derivation details are in the companion technical report [21]):

$$\frac{d(-Exe\_time)}{dSize_{absolute}} \geq \frac{k * Exe\_time}{IPC_{static} * IC_{static}} \quad \textbf{(4)}$$

In Equation 4, IC_{static} represents the static operation count of the program (i.e., the static code size), *k* is the threshold on instantaneous code size efficiency and the term *d*(-*Exe_time*) represents the decrease in the execution time. The terms *Exe_time* and IPC_{static} represent the global features of the program. In this paper, the execution time and IPC based on natural treegion scheduling shown in Table 3 are used. Now, the algorithm to find the best code size efficiency is a simple threshold approach, as shown in Figure 10.

Algorithm for finding the best code efficiency based on tail duplications

0. Mark the loop edges so that the tail duplication will not overlap with cyclic optimization such as loop unrolling and calculate the threshold using Equation 4 with k setting to anywhere between tan(π/6) to tan(π/12).
1. Calculate the instantaneous code size efficiency for all possible tail duplication candidates in the program scope.
2. If there is a candidate whose instantaneous code size efficiency is above the threshold, duplicate the candidate and update the efficiency of affected candidates, repeat until there are no more candidates.

Figure 10. Algorithm for finding the best code size efficiency based on tail duplication

As the threshold *k* represents the slope of tangent line around the best solution point, one reasonable range for *k* is from tan(π/6) to tan(π/12) as the corresponding tangent lines will hit the points close to the knee of the curve. For example, if we choose *k* as 0.577 (corresponding to the case that the tangent line at optimal point has the angle of π/6) for benchmark *vortex*, the threshold becomes 1820, which means that if the tail

Table 5. The experimental results for threshold k = 0.577

Benchmark	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
Efficiency threshold	3354	467	1543	3657	2436	625	3417	1820
Resulting Relative Code Size	1.09	1.024	1.06	0.998	1.0	1.0	0.969	1.027
Resulting IPC	2.76	2.71	2.165	2.734	2.487	2.278	2.895	3.416
IPC (20% code size increase)	2.79	2.73	2.206	2.745	2.492	2.300	2.910	3.444

Table 6. The experimental results for threshold k = 0.268

Benchmark	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
Efficiency threshold	1561	217	716	1698	1131	290	1587	846
Resulting Relative Code Size	1.13	1.05	1.11	1.006	1.003	1.01	0.972	1.045
Resulting IPC	2.78	2.72	2.192	2.739	2.489	2.285	2.898	3.427

duplication candidate can result in more than 1820 cycles speedup at cost of 1 additional operation, then this tail treegion should be duplicated. The thresholds calculated for all the benchmarks and the resulting (static) IPC and code size combinations after treegion scheduling are shown in Table 5. The IPC resulting from 20% code size increase is also included in the table.

From the results in Table 5, it can be seen that the benchmarks can be grouped into three categories. The first category has the feature that the code size efficiency reaches the 'diminishing returns' very soon (i.e., the resulted code size is same or less than the original code size while the static IPC almost reaches the maximum). Benchmarks *ijpeg*, *li*, *m88ksim* and *perl* belong to this category. For the second category benchmarks including *gcc* and *vortex*, such diminishing returns happen with a relatively small increase from the original code size (2.4% and 2.7% respectively for gcc and vortex). The other two benchmarks *compress* and *go* are in the third category, which require more code size increase to reach the maximal IPC.

If we change the threshold on instantaneous code size efficiency to 0.268 (corresponding to the case that the tangent line at optimal point has the angle of $\pi/12$), the calculated thresholds, the resulting IPC and code size combinations after treegion scheduling are shown in Table 6. As expected, for benchmarks in first and second category, the variation in *k* results in very small change in the results. For benchmarks in the third category, such variation results in around 5% change in code size and 1% in performance, which, in our opinion, are still valid solutions for optimal code efficiency.

Here, we pick one benchmark in each category to show graphically where the points are found with the threshold scheme. The benchmark *m88ksim* is picked from the first category and its IPC vs. code size curve is shown in Figure 11 using the best IPC results for given code size increase for 0%, 2%, 5%, 10% and 20%. From Figure 11, it can be seen that the threshold scheme locates the optimal point accurately. Benchmarks *vortex* and *compress* are chosen from the second category and the third category respectively and their IPC vs. code size curve can be seen in Figure 5 and 6. From those figures, we can conclude that this simple threshold scheme finds the best efficiency solutions accurately.

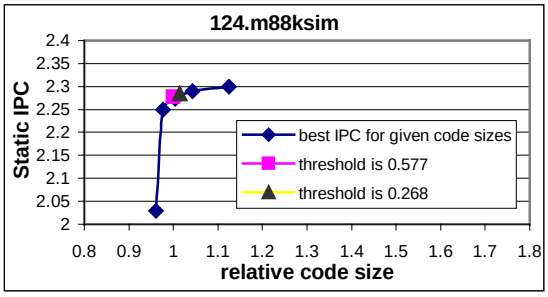


Figure 11. The best code size efficiency found using different thresholds for benchmark *m88ksim*

To investigate the associated I-cache performance due to the code size increase, a medium-sized I-cache (32KB as specified in Table 1) is used in the detailed timing simulation. In this experiment, we compare the I-cache performance of natural treegion results to the optimal efficiency results obtained with threshold as 0.577. Figure 12 shows the I-cache miss rates of each benchmark for these two cases.

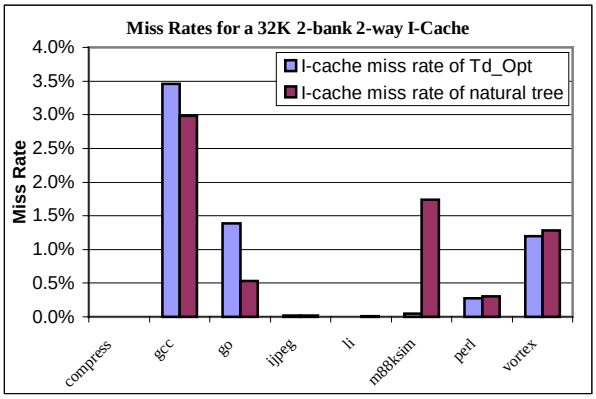


Figure 12. I-cache miss rates for natural treegion and the optimal efficiency results obtained with threshold as 0.577

In Figure 12, benchmarks *gcc* and *go* show significant increases in I-cache miss rate due to the code size increase of the optimal efficiency results while other benchmarks exhibit similar or smaller I-cache miss rates. The reason for the decreases in I-cache miss rates is mainly due to the effect that the tail duplication in optimal efficiency results increases the sequential locality of the frequently executed regions, as observed

in [3]. Another fact that improves the I-cache performance is that the tail duplication enables the treegion scheduler to produce a denser schedule of the operations (i.e., more operations in each multi-op). As a result, the number of I-cache accesses is reduced and so is the number of I-cache misses. Figure 13 shows the ratio of I-cache misses of the optimal efficiency results to the natural treegion results. It can be seen from Figure 12 and 13 that although the optimal efficiency results of the benchmark *gcc* has a higher miss rate than natural treegion results, it has smaller I-cache miss penalties due to the reduced number of accesses. In average, the I-cache miss penalties of optimal efficiency results have a 4% decrease comparing to the natural treegion results for a 32KB I-cache.

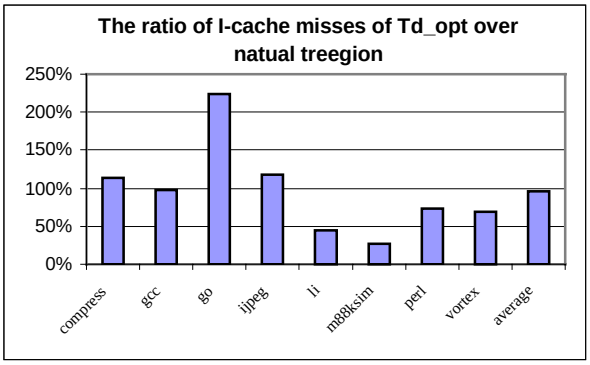


Figure 13. The ratio of I-cache misses of optimal efficiency results over natural treegion results

Overall, in Figure 14, we show the performance with realistic I-cache, D-cache, and branch prediction (the parameters are in Table 1) and the ideal performance assuming ideal cache and branch prediction (i.e., the static IPC) for treegions formed using optimal code size efficiency, Harvanki's heuristic, and natural trees. From Figure 14, it can be seen that the optimal efficiency results show an average of 22% speedup based on static IPC and 17% speedup based on dynamic IPC over natural treegion results. In terms of the code size increase, natural treegion results, Havanki's results and optimal efficiency results show an increase of –3%, 70%, and 2% over the original code size respectively.

5. Conclusion

This paper presents a code size efficiency study for global scheduling for ILP processors. The main contributions include:

- A *quantitative measure* of *the code size efficiency* is proposed for any code size related optimization. Based on the general idea of expressing the code size efficiency as the ratio of IPC changes over the code size changes, two formal definitions are formulated, *the average code size efficiency* and *the instantaneous code size efficiency*, and they are used to measure the average impact of code size related optimizations and the effect of an individual application of an optimization respectively.
- A heuristic based on performance bound is proposed to estimate the execution time of a multi-path region so that we can convert the static IPC computation in code size efficiency into the estimated execution time.
- We proposed an *iterative approach* to find *the best code size efficiency for a given code size constraint*. Using the tail duplication as an exemplary code size related optimization, it is shown that code size increase resulting from tail duplication has a significant but varying impact on IPC, e.g., the first 2% code size increase results in 18.5% increase in IPC while the IPC changes less than 1% when given code size increase ranging from 20% to 30%.
- Based on the observations made above, we define the term of *optimal code size efficiency for any program* and a simple, yet robust threshold

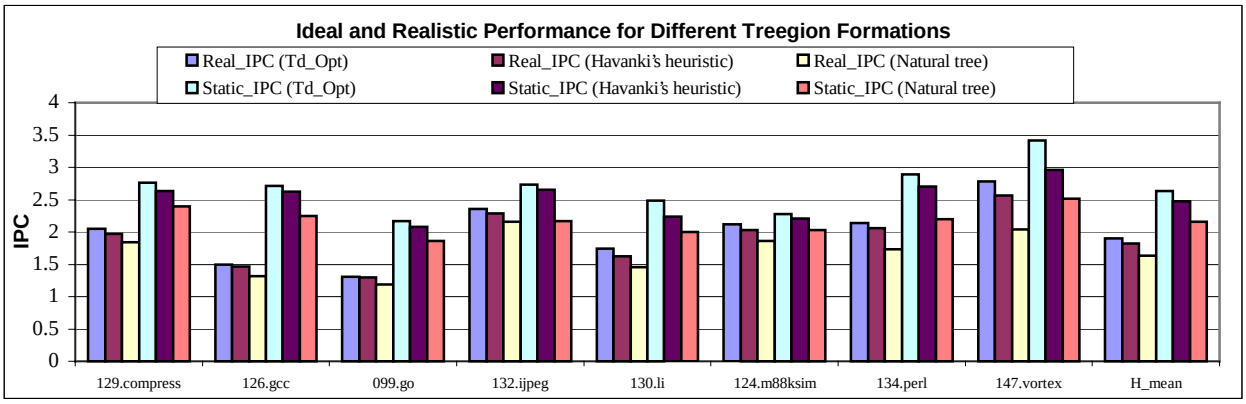


Figure 14. The ideal and realistic performance for different treegion formations

scheme is derived to find this optimal solution. Our experimental results verified that this scheme finds the optimal code size efficiency accurately and for SPEC95int benchmarks, it shows average of 2% code size increase of scheduled code over the original code and improved I-cache performance (4%) for a medium size cache (32K) comparing to the natural treegion scheduled results. In terms of performance, the optimal efficiency results show an average of 22% based on static IPC and 17% speedup based on dynamic IPC over natural treegion results. So, with a small code size increase, significant ILP can be better exploited during the global scheduling phase while the I-cache performance is improved at the same time.

The code size efficiency enables us to find the best trade-off between static ILP exploitation and code size increase. We can extend this approach for different code size related optimizations. For example, we may use the efficiency to decide whether to unroll a loop for a certain times or to tail duplicate one candidate region.

6. Acknowledgments

This research was funded by Intel Corporation and Sun Microsystems, Inc.

7. References

[1] W.A. Havanki, S. Banerjia, and T. M. Conte. "Treegion scheduling for wide-issue processors." *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[2] H. Zhou, M. Jennings, and T. M. Conte. "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors". *Proceedings of the* 14th *Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01), LNCS, Springer Verlag*, August 2001.

[3] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design", Technical Report CRHC-91-17, University of Illinois, Urbana, May 1991

[4] M. Jennings, H. Zhou, and T. M. Conte. "A Treegion-based Unified Approach to Speculation and Predication in Global Instruction Scheduling". Technical Report, ECE Department, NC State University, August 2001.

[5] 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 way for VLIW and superblock compilation." *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

[6] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: version 1.1." Tech. Rep. HPL-93-80 (R.1), Hewlett--Packard Laboratories, February 2000.

[7] 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" *Proc. 25th Ann. Int'l Symp. Microarchitecture (MICRO25)*, December 1992.

[8] S. Aditya, V. Kathail, and B. R. Rau, "Elcor's machine description system: version 3.0." Tech. Rep. HPL-98-128 (R.1), Hewlett--Packard Laboratories, October 1998.

[9] M. S. Schlansker and B. R. Rau. "EPIC: An architecture for instruction-level parallel processors" Tech. Rep. HPL-99-111, Hewlett--Packard Laboratories, February 2000.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors", *Proc. 18th Int'l Symp. On Computer Architecture (ISCA18)*, 1991.

[11] The LEGO Compiler. Available for download at http://www.tinker.ncsu.edu/LEGO.

[12] F. Mueller and D. B. Whalley, "Avoiding Conditional Branches via Code Replication", ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun 1995.

[13] D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling", *Proc. 24th Ann. Int'l Symp. Microarchitecture (MICRO24)*, 1991.

[14] Bill Mangione-Smith, "Performance Bounds for Rapid Computer System Evaluation", in Fast Simulation of Computer Architectures, edited by Thomas M. Conte and Charles E. Gimarc, Kluwer Academic Publishers, 1995.

[15] Intel Corp, IA-64 Application Developer's Architecture Guide, 2000.

[16] B. R. Rau, "Iterative Module Scheduling", Tech. Rep. HPL-94-115, Hewlett--Packard Laboratories, 1995.

[17] A. E. Eichenberger and W. M. Meleis, "Balance Scheduling: Weighting Branch Tradeoffs in Superblocks", *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO32)*, 1999.

[18] Rainer Leupers, "Code Optimization Techniques for Embedded Processors", Kluwer Academic Publishers, 2000.

[19] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings." *Proc. 29th Ann. Int'l Symp. Microarchitecture (MICRO29)*, December, 1996

[20] J. Hoogerbrugge. "Dynamic branch prediction for a VLIW processor." *Proc. Of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT'00)*, October 1997.

[21] H. Zhou and T. Conte. "Code Size Efficiency in Global Scheduling". Technical Report, ECE Department, NC State University, January 2002.

Code Compression by Register Operand Dependency

Kelvin Lin, Jean Jyh-Jiun Shann and Chung-Ping Chung
Department of Computer Science & Information Engineering
National Chiao Tung University
{kelvin, jjshann, cpchung}@csie.nctu.edu.tw

Abstract

This paper proposes a dictionary-based code compression technique that maps the source register operands to the nearest occurrence of a destination register in the predecessor instructions. The key idea is that most destination registers have a great possibility to be used as source registers in the following instructions. The dependent registers can be removed from the dictionary if this information can be specified otherwise. As a result, the compression ratio benefits from the decreased dictionary size. A set of programs has been compressed using this feature. The compression results show that the average compression ratio is reduced to 38.6% on average for MediaBench benchmarks compiled for MIPS R2000 processor.

1. Introduction

Most of the embedded systems are cost sensitive. Small memory size results in a lower cost and lower power requirement. Typically, programs in an embedded system are stored in a ROM associated with an ASIC, whose sizes translate directly into silicon area and cost. Thus, memory size reduction becomes more important in the design of an embedded system. In addition, as the complexity of an embedded system grows, programming in assembly language and optimization by hands are no longer practical and economical. The programs are written in high-level languages (HLL), such as C and C++, and compiled into executables. Direct translation from high-level languages into the machine code incurs the penalty of code size due to completeness of translation for each HLL statement to machine instructions. Some code optimization, such as redundant code removal or common sub-expression elimination must be extra processed [1]. Most compiler optimizations focus on the execution speed rather than the code size, and this fact results in a speed-space trade-off. Therefore, code size optimization has great potential within the described a programming environment.

This paper proposes a code compression technique that further reduces code size. This method is based on the operand factorization [2], but separates the instruction sequence differently into the opcode sequence, the mapping sequence, and the residual operand sequence. The key idea of this method is that a source register has a great possibility of congruence with the destination

register of the previous instruction. We use the mapping tag to identify the relationships between source registers and destination registers so that the occurrences of the same registers can be eliminated from the operand sequences used in the operand factorization method. We found that the variations of the relations are much smaller than that of the operands themselves. The dictionary storing the mapping information occupies only a small amount of space, and the size of the dictionary storing the operands is greatly reduced. As a result, compression ratio benefits from the decreased dictionary size. Experimental results show that the compression ratio can be reduced to 38.6% on average for MediaBench [3] compiled for MIPS R2000 processor.

This paper is organized as follows: Section 2 discusses the related work in code compression; Section 3 proposes the detailed register dependency method; Section 4 describes the decompression engine; Section 5 presents the simulation results, and Section 6 is a summary.

2. Related Work

An intuitive way to achieve the reduction of codes is to restrict the instruction size. This is the approach adopted in the design of the Thumb [4] and MIPS16 [5]. Shorter instructions are achieved primarily by restricting the number of bits that encode opcodes, registers and immediate values. The results are 30%~40% smaller programs running 15%~20% slower than programs using standard RISC instruction set.

Another way to reduce the code size is to use the traditional compression method, which encodes the occurrences of identical instructions (or instruction sequences) in a program to the smaller codewords to reduce the program size. Lefurgy et al. [6] propose a dictionary-based compression method, which stores one copy of the common instruction sequences into the dictionary and replaces the occurrences of the sequences with shorter (fixed or variable-length) codewords than the instruction sequences themselves. Post-compilation modifies all branch offsets to reflect the new compressed address space. The average compression ratios of 61%, 66%, and 74% were reported for the PowerPC, ARM, and i386 processors respectively. Wolfe el al. [7] propose a statistical compression method in Compressed Code RISC Processor (CCRP). Each 32-byte cache line is Huffman-encoded [8] into smaller aligned bytes or words. Line Address Table (LAT) generated by the compression tool is used to map the original program instruction addresses into compressed code instruction addresses. This table is stored along with the program. The compression ratio of 73% on MIPS R2000 was reported.

To further improve the compression ratio, more similarities between instructions must be explored to reduce both the dictionary size and the program encoding. Araujo et al. [2] find that most instruction sequences are identical with either opcode sequences or operand sequences, but not both. Therefore, they separate the instruction sequences into tree-patterns (opcode sequences) and operand patterns (operand sequences) and encode each instruction sequence is encoded into T_P (tree pattern codeword) and O_P (operand pattern codeword). This method is called operand factorization. The average compression ratio for this scheme is 43% using Huffman encoding [8]. Another coding instruction steam method, called tailored encoding [9], tries to minimize both the encoding of opcodes and instruction sizes. This method minimizes the number of bits to encode the opcodes and registers exactly used in an application, and shortens the instruction length by omitting the reserved field or

narrowing the storage of an immediate value. In the end, the compact instructions are considered as basic unit to be compressed into Huffman-codes. An average compression ratio of 65% was reported.

Encoding the instruction stream utilizing register dependency is found in early Horizon machine [10]. Each 64-bit instruction contains a lookahead field that is used to control the instruction overlap. This field specifies the number of instructions without register dependency. This field generated by the compiler is used primarily to denote the maximal number of instructions that can be issued before a dependency is encountered to maximize the instruction level parallelism. Register dependencies are also utilized to minimize the memory traffic during procedure calls [11]. This method proposes four compiler optimizations guided from profile information to eliminate the load/store of callee-saved register during the procedure calls. An average of 2.5% speedup is obtained.

3. Register Operand Dependency

The following subsections detail our compression method. First of all, we identify the register operands in the instructions. Observation shows that the instruction sequences have dependencies between register operands, so we find out the destination-source dependencies between them. Finally, we compress the instruction sequences into Huffman-codes. The following procedures describe the compression algorithm using such a technique.

3.1. Instruction Classification

This step examines the types of operands in the instruction formats to find the register dependency. To build the compression model, we examine the instruction set of MIPS R2000 processor [12]. The instructions are classified into the following categories shown in Table 1 according to the types of operands in the instructions. In this table, *op* indicates the opcode of an instruction, *src* (*dst*) indicates the operand is used as a source (destination) register for operation *op*, and *imm* indicates the field is an immediate. This classification is based on the register types and the number of registers the opcode exactly used. The instructions ‘nop’ and ‘move’ are pseudo instructions, but not assembly instructions. The pseudo instruction ‘move $r*n*, $r*m*’ can be implemented (generated by GNU GCC) by either instruction ‘addu $r*n*, $r*m*, $r*0*’ or ‘or $r*n*, $r*m*, $r*0*’. The opcode ‘addu’ with some specified operands, such as ‘$rn, $rm, $r0’, is encoded into a new opcode. We believe that encoding such an opcode with a restricted operand distinctly from the original opcode reduces code size more than encoding of all combinations of possible operands. There are two reasons: First, since there are only a small number of distinct opcodes for MIPS R2000, encoding opcode with restricted operand

Table 1. Instructions classification

Categories	Example Instruction
1. op	*nop
2. op src	mthi $r*n*
3. op dst	mfhi $r*n*
4. op imm	j *address*
5. op src, src	mult $r*n*, $r*m*
6. op src, imm	bgez $r*n*, *address*
7. op dst, src	*move $r*n*, $r*m*
8. op dst, imm	lhi $r*n*, *value*
9. op src, src, imm	sw $r*n*, *offset*($r*m*)
10. op dst, src, src	add $r*n*, $r*m*, $r*k*
11. op dst, src, imm	lw $r*n*, *offset*($r*m*)

into a new opcode increases fewer bits than the encoding for all combinations of possible operands. Second, some opcodes are likely to use specified operands. Encoding such a case into a distinct opcode can shorten the operand sequences. Shorter operand sequences are more likely to be shared by multiple instruction sequences. This classification is used to find the dependency relationships between source and destination registers.

3.2. Register Operand Dependency

Observation reveals that instruction sequences may have the same dependency relationship between registers even if they are different instruction sequences. Consider the following two instruction sequences and their opcode sequences and operand sequences extracted according to the instruction format in Figure 1. Both second

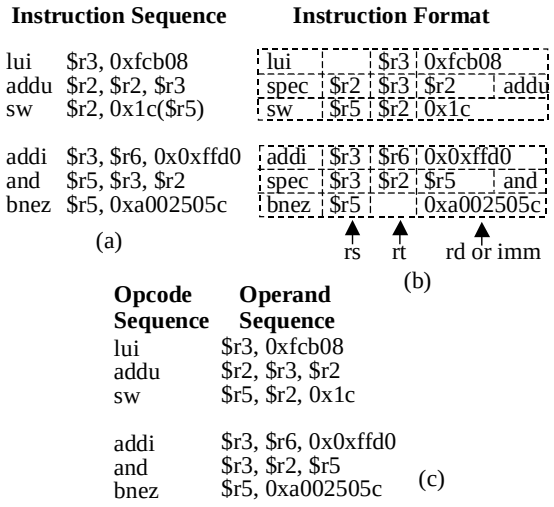


Figure 1. Example opcode sequences and operand sequences

instructions of these two sequences use the destination registers of the previous instructions as their source registers. A *mapping tag* is used to describe this dependency. Each source register has a mapping tag. For an *n*-bit mapping tag, we assign the value '*0*' for *load operation* indicating that the corresponding register must be retrieved from the *Residual Operands*, and value '*k*' ranging from *1* to 2^n*-1* for *relation operation* indicating that the corresponding register can be obtained from the destination register of the previous instructions. The mapping distance, indicated by a non-zero value, counts only the number of instructions with a destination register, from the instruction writing to this source register of the current instruction. The remaining unmapable registers and the immediate values are placed behind as the residual operands. We use a *2*-bit mapping tag for example. Tag value '*0*' indicates the corresponding source register comes from residual operands, and tag values '1', '2' and '3' indicate the corresponding source register is the destination register of the previous first, second and third instruction, respectively. Figure 2 shows the representation of relationships and the residual operands.

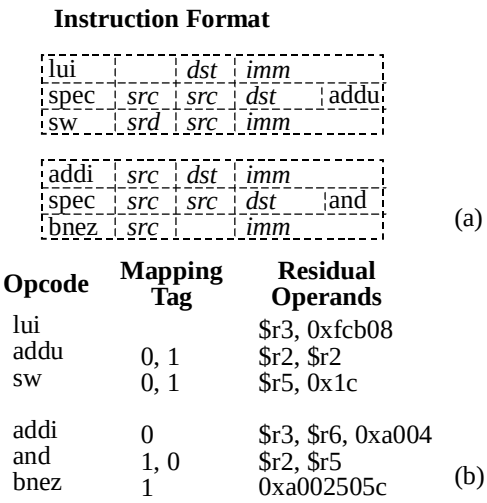


Figure 2. Representations of operand relationships and residual operands

The first instruction does not contain any source register, so there is no need for any mapping tag. All the operands are listed as the residual operands. In the second instruction, the first source register ($r2) has not appeared before, so the mapping tag is '*0*' and this register is placed

behind as the residual operand. The second source register ($r3) is the same as the destination register of the first instruction, and thus a tag value '*1*' indicates that this source register is the same as the destination register of the instruction one position up. As a result, this register can be omitted. The destination register is also placed behind as the residual operands. The other instructions proceed in like manner. It is surprising that these two instruction sequences in the example have the same mapping tag sequence although they are quite different instructions. This method not only extracts the common register relations, but also reduces the number of operands in the operand sequence to increase the re-usage of the residual operands. This step attempts to locate the register dependencies for all instructions in the basic blocks. These dependency relationships cannot cross the basic block boundary since the registers are not guaranteed to be alive across the basic block.

3.3. Register Majority

After removing the dependent source registers from operand sequence, the residual operand sequences still contain redundancies. There are some registers appearing frequently in residual operand sequences. The most frequent register among all residual operand sequences is termed the first (register) majority, the second most frequent register is termed the second majority, and so on. An effective method to further reduce the dictionary size is to remove these majorities. We borrowed values from the mapping tag to denote the mapping of the unmappable registers to these majorities. Assuming that there are *m* majorities, for an *n*-bit mapping tag, the tag value '*0*' is reserved for load operation, the tag values from '*1*' to '$2^n - m - 1$' remain to the relation operation and tag values from '$2^n - m$' to '$2^n - 1$' indicate these *m* majorities. The majorities are stored in dedicated registers, called *Majority Registers* (MRs), so that the mapping tag can reference these majorities. For example, assuming that the register '$r5' is the first and the only majority for a *2*-bit mapping tag, tag value '3' is reserved for majority. The instruction sequences in Figure 2 can be transformed into Figure 3. This step replaces the tag value '*0*' with value '*3*' if the corresponding register is a majority, and removes the majority registers from the residual operand sequences.

Mapping Tag	Residual Operands
	$r3, 0xfcb08
0, 1	$r2, $r2
0, 1	$r5, 0x1c
0	$r3, $r6, 0xa004
1, 0	$r2, $r5
1	0xa002505c

⇒

Mapping Tag	Residual Operands
	$r3, 0xfcb08
0, 1	$r2, $r2
3, 1	0x1c
0	$r3, $r6, 0xa004
1, 3	$r2
1	0xa002505c

Figure 3. Mapping tag sequence after applying the majorities

3.4. Program Encoding

After removing the registers with destination-source and majority relationships from the operand sequences, the program is divided into instruction sequences as the basic unit for compression. An instruction sequence is defined as a number of instructions with each instruction (except the first one) having at least one source register to be the same as the destination register of the predecessor instructions. Every instruction sequence is partitioned into three sequences: opcode sequence, mapping tag sequence and residual operand sequence. These three sequences are independently Huffman-coded into codewords CW_{OP}, CW_{MAP} and CW_{OD}. The entire program is transformed into the form: [CW_{OP1}, CW_{MAP1}, CW_{OD1}, CW_{OP2}, CW_{MAP2}, CW_{OD2}, …, CW_{OPi}, CW_{MAPi}, CW_{ODi}], assuming that the

program has a total of *i* instruction sequences. Codewords are allowed to split at the end of bytes. Bits from the spilled codeword are spilled into the next byte. The compression ratio benefits from the use of splitting codewords and Huffman encoding. But doing so also cause the execution overhead. We trade-off the performance in exchange for more compression ratio.

3.5. Branch Target Address

Since the codewords can be of any length and not necessarily byte aligned, the branch target must be able to point at any bit location within a byte. The branch offset is divided into 2 fields: the byte address (23 bits or 13 bits) and the bit offset (3 bits). The overall branch distance is reduced to 1/32. Nevertheless, for the program analyzed, only a small percentage of targets require more than 23 or 13 bits. For those branches, a jump table is provided for storing the target addresses. Similar to [6], the jump table addresses are patched up to reflect the compressed addresses.

4. Decompression Engine

This section describes the decompression engine designed for our compression method. The decompression engine consists of three dictionaries and an *Instruction Assembly Buffer* (IAB) as shown in Figure 4.

4.1. Dictionaries

Three dictionaries store the opcode sequences, mapping tag sequences and residual operand sequences, independently. The opcode sequences are stored in the *Opcode Dictionary* (OPD). Each entry contains 3 fields:

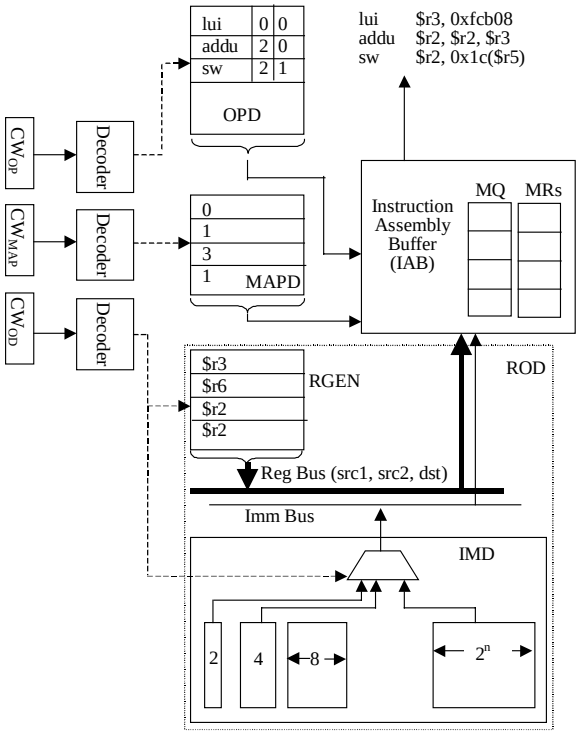


Figure 4. Decompression engine

the *opcode* field, the *mapping type* field and the *end mark*. The opcode field contains the opcode bits of an instruction and the mapping type field contains two bits indicating how many mapping tags must be read for this instruction. The opcodes with different operand sequences will be encoded into different OPD entries due to their different mapping type fields. The end mark signals the end of an instruction sequence.

The mapping tag sequences are stored in the *Mapping Dictionary* (MAPD). We optimize the MAPD size by sharing a shorter mapping tag sequence with a longer one of which the prefix sub-sequence is the same as the shorter one.

The *Residual Operand Dictionary* (ROD) storing the residual operand sequences consists of two storages, the *Register Generation* (RGEN) and *IMmediate value Dictionary* (IMD). As the method proposed by Araujo [2], the RGEN stores the registers only and the IMD stores

the immediate values in the residual operand sequences. Residual operand sequence with immediate values can be used to minimize the RGEN. For example, the residual operand sequences '$r4, $r5, 0x4' and '$r4, $r5, $r2' can share the same register sequence '$r4, $r5, $r2'. The last register is also sent to the IAB from *Register Bus* (RegBus), but ignored by IAB according to the opcode from OPD. By rearranging the ROD to RGEN and the IMD, the residual operand sequences of the two instruction sequences in Figure 3 can share the same entry in RGEN. On the other hand, IMD stores each distinct immediate value in the program, regardless of which residual operand sequence contains it. These values are clustered into memory banks according to the number of bits consumed. Accessing IMD and RGEN can be processed in parallel to accelerate the decompression.

4.2. Instruction Assembly Buffer

The CW_{OP}, CW_{MAP} and CW_{OD} are extracted from the compressed program to index to instruction opcodes, mapping tags and residual operands, respectively. The retrieved opcodes, mapping tags and residual operands are sent to the instruction assembly buffer (IAB) to assemble to the original instruction sequences. Every time an instruction is assembled, the destination register (if any) is pushed into the *mapping queue* (MQ), so that the mapping tag can reference them when relation operation is specified. The size of the MQ is equal to the maximal mapping distance defined previously.

4.3. Discussion on Decompression Overhead

The major concern for register operand dependency method is the decompression efficiency. The decompression speed depends on three periods of time:

(a) Determining the lengths of three codewords (CW_{OP}, CW_{MAP} and CW_{OD}).
(b) Decoding of each codewords, and
(c) Assembling the instruction sequence.

Speeding the determination of the lengths of codewords and decoding of the codewords, which also happens to the operand factorization method, can be solved by parallel Huffman decoder [13]. The critical path for assembling an instruction sequence differs from one sequence to another. The mapping tags are read sequentially depending on the mapping types in OPD entry. The more mapping tags to be read, the slower the decompression speed. Furthermore, after assembling an instruction, the destination register must be pushed into MQ, which is also manipulated serially. These two steps are the main penalty for exchange of a better compression ratio.

5. Experimental Results

This section describes the experimental results of code compression by register dependency. We use the MediaBench [3] for analyzing this technique. The programs are compiled for MIPS R2000 using GCC version 2.8.1 with optimization –O2. Initially, we examine the suitable size of mapping tag to find the best compression ratio, and then compare the compression ratios between this method and the operand factorization method. The following subsections explain the compression effects using the cjpeg for example.

5.1. Size of Mapping Tag

It is critical to determine what size of mapping tag is sufficient for compacting the both dictionaries and the

program encoding. Simulation is used to find the suitable tag size. The simulated tag sizes range from *1* to *5* bits (since the encoding of register in the original program is 5 bits), and the number of majorities ranges from *0* to $2^{tag\ size} - 1$. The tag value '*0*' is always for load operation, tag value '*1*' to '*k*' ($k = 2^{tag\ size} - \#\ majorities - 1$) indicates the relation operation and tag value '$k+1$' to '$2^{tag\ size} - 1$' is for the majority operation.

5.2. RGEN Size Reduction

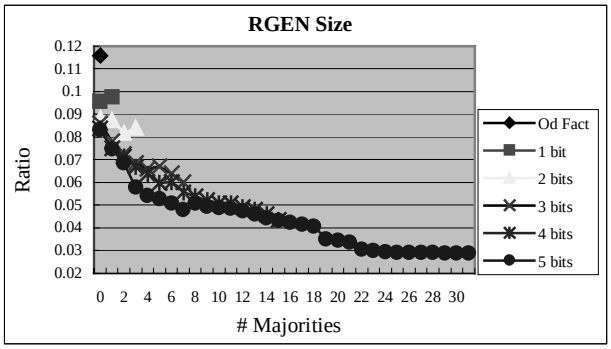


Figure 5. RGEN Size vs. mapping tag size

Figure 5 shows the dictionary size reduction versus the size of mapping tag for RGEN. The x-axis is the number of majorities and the y-axis is the RGEN size ratio to the original program size. The first data set consists of one point showing the size ratio of RGEN resulting from the operand factorization method. The second data set is a line-graph consisting of two points showing 1-bit mapping tag with zero and one majorities. The third data set is also a line-graph consisting of four points showing a 2-bit mapping tag with zero, one, two and three majorities, respectively. From the second data set (n = 1 case), we see that size reduction due to dependency is more than due to the removal of majorities. Furthermore, the last point of each curve always tilts up. This is because the number of registers removed from residual operands by mapping distance 1 is sufficiently larger than the number of $(2^n-1)^{th}$ majority. From this figure, a 5-bit mapping tag with 30 majorities reduces the RGEN size by the largest degree.

5.3. Mapping Penalty

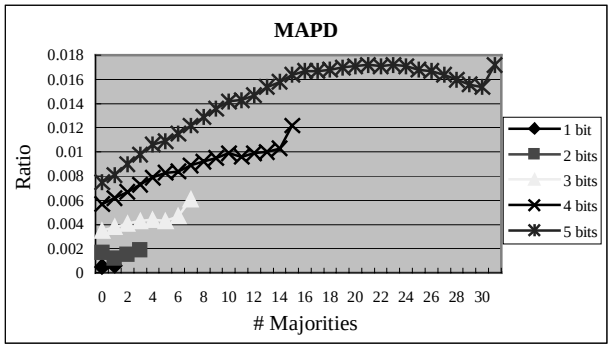


Figure 6. MAPD size vs. mapping tag size

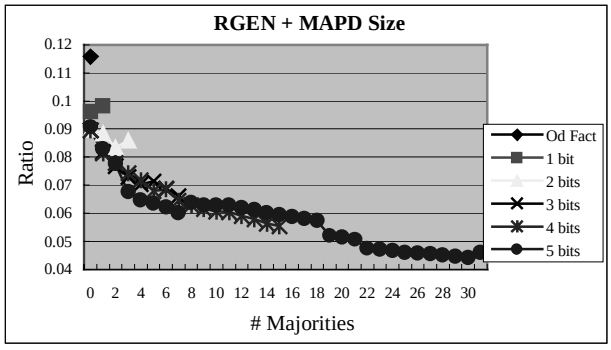


Figure 7. Size of RGEN plus MAPD vs. mapping tag size

Although the size of RGEN is reduced, we introduced the MAPD. Figure 6 shows the MAPD size compared to the original program. The x-axis is the number of majorities and the y-axis is the MAPD size ratio. Fortunately, the size of MAPD is much smaller than the size reduced in RGEN. The overall effect is still positive for compression. Figure 7 shows the size of MAPD plus RGEN. The overall dictionary size reduction is 5.41% on average by using register dependency compression method.

5.4. Final Compression Ratio

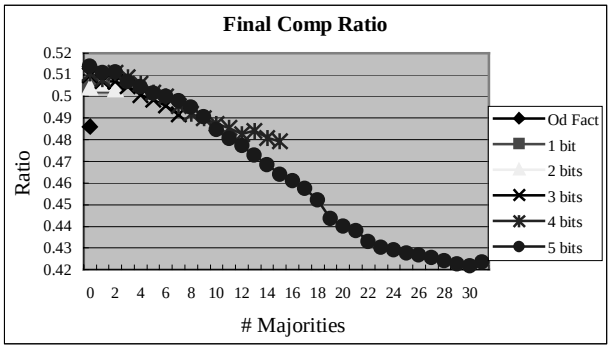


Figure 8. Final compression ratio vs. mapping tag size

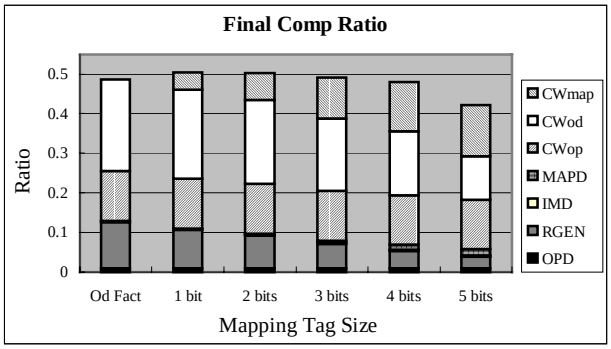


Figure 9. Size contributions for all components in a compressed program

Figure 8 shows the final compression ratios versus the size of mapping tag. As we expected, a 5-bit mapping tag with 30 majorities results in the best compression ratio. Figure 9 shows size contributions of the components in the compression ratio. The x-axis shows the mapping tag size and the y-axis shows the best compression ratios of the specified mapping tag sizes. This figure shows that three large portions in a compressed program are CW_{OP}, CW_{MAP} and CW_{OD}. Table 2 shows size reductions when the components are classified into two major parts: dictionary part and program encoding part. Dictionary part indicates the total size of OPD, MAPD, RGEN and IMD. The program encoding consists of CW_{OP}, CW_{MAP} and CW_{OD} of all instruction sequences. From Table 2, the maximal factor in reducing the compression ratio is due to the reduction of dictionary size rather than encoded program size.

Table 2. Size reduction of dictionary and program encoding

Tag Size	Dictionary Size(%)	Compressed Code Size(%)	Dictionary Size Reduction(%)	Program Encoding Reduction(%)
Operand Factorization	12.91	35.69	N/A	N/A
1-bit	10.96	39.45	2.00	-3.80
2-bits	9.66	40.67	3.20	-5.00
3-bit	7.94	41.21	5.02	-5.50
4-bits	6.89	41.04	6.00	-5.40
5-bits	5.76	36.4	7.20	-0.70

Figure 10 shows the final compression ratios for all benchmark programs. Each benchmark consists of 2 bars, one for operand factorization (Od Fact) method and the other for our register operand dependency (Reg Dep) method. The OPD, IMD and CW_{OP} are the invariants in these two methods. The average decrement of the RGEN is 6.6% and the increment of MAPD is 1.2%. This is the main advantage of register dependency method. The average decrement of CW_{OD} is 8.3%, but the increment of CW_{MAP} is 10.2%. Total detail statistics are given in Table 3.

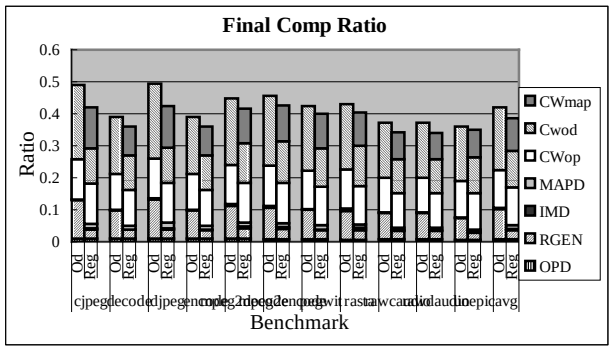


Figure 10. Final compression ratio for all benchmarks

Table 3. Final compression ratios

Benchmark	Method	OPD (%)	IMD (%)	RGEN (%)	MAPD (%)	CW_{OP} (%)	CW_{OD} (%)	CW_{MAP} (%)	Final (%)
cjpeg	Od Fact	1.00	0.33	11.58		12.55	23.14		48.60
	RegDep	1.00	0.33	2.88	1.54	12.55	10.99	12.86	42.15
decode	Od Fact	1.06	0.16	8.77		11.19	17.96		39.14
	RegDep	1.06	0.16	2.71	1.06	11.19	10.88	8.96	36.02
djpeg	Od Fact	1.07	0.26	12.25		12.52	23.44		49.54
	RegDep	1.07	0.26	2.85	1.78	12.52	10.89	13.03	42.40
encode	Od Fact	1.05	0.17	8.75		11.19	17.96		39.12
	RegDep	1.05	0.17	2.70	1.05	11.19	10.87	8.96	35.99
mpeg2decode	Od Fact	1.05	0.53	10.21		12.26	20.82		44.87
	RegDep	1.05	0.53	3.24	1.32	12.26	12.47	10.73	41.60
mpeg2encode	Od Fact	0.89	0.66	9.78		12.53	21.73		45.59
	RegDep	0.89	0.66	3.18	1.18	12.53	13.09	11.15	42.68
pegwit	Od Fact	0.83	0.14	9.32		11.92	20.28		42.49
	RegDep	0.83	0.14	2.84	1.44	11.92	12.09	10.86	40.12
rasta	Od Fact	0.71	0.72	9.02		12.13	20.41		42.99
	RegDep	0.71	0.72	2.96	1.02	12.13	12.53	10.32	40.39
rawcaudio	Od Fact	0.91	0.14	8.20		10.78	17.19		37.22
	RegDep	0.91	0.14	2.51	0.98	10.78	10.50	8.34	34.16
rawdaudio	Od Fact	0.91	0.14	8.19		10.77	17.18		37.19
	RegDep	0.91	0.14	2.51	0.98	10.77	10.50	8.34	34.15
unepic	Od Fact	0.68	0.31	6.74		11.34	17.02		36.09
	RegDep	0.68	0.31	2.20	0.7	11.34	11.29	8.47	34.99
Agv	Od Fact	0.92	0.32	9.38		11.74	19.75		42.12
	RegDep	0.92	0.32	2.78	1.19	11.74	11.46	10.18	38.60

6. Conclusion

In this paper, we propose the register dependency compression method to compress embedded system programs for a RISC processor. The key idea of this method is to remove the dependent register from the operand sequences to reduce the dictionary size and program encoding. The best compression ratio of this method results in 34.15% and an average of 38.60%.

This research can be further improved in several ways. First, the codewords for both opcode sequences and mapping tag sequences are the largest portions contributing to the compression ratio. Reducing mapping tag size and reusing the OPD entries are next step goals for improving the compression ratio. Second, the compiler could attempt to produce identical instruction sequences for the same expression tree [1] so that the more common instruction sequences become more compressible [14]. One way to accomplish this is to allocate the same registers for the same expression tree. Finally, we can improve the algorithm to find more relationships between operands. Such implementation may include building both the language grammar and register allocation rules, and compressing the instruction sequences to the representations of these rules.

7. References:

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, Boston, 1988.

[2] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization," *31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[3] C. Lee, M. Potkonjak, and W. H. M. Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[4] J. L. Turley, "Thumb Squeezes ARM Code Size," *Microprocessor Report*, 9(4), 27 March 1995.

[5] K. Kissell, *MIPS16: High-density MIPS for the Embedded Market*, Silicon Graphics MIPS Group, 1997.

[6] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," *Proceedings of the 30th Annual International Symposium on*

Microarchitecture, December 1997.

[7] A. Wolfe and A. Chanin, “Executing Compressed Programs on an Embedded RISC Architecture,” *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[8] D. A. Huffman, “A Method for the Construction of Minimum Redundancy Codes,” *Proceedings of the IEEE*, vol. 40, 1952, pp. 1089 – 1101.

[9] S. Y. Larin and T. M. Conte, “Compiler-driven cached code compression schemes for embedded ILP processors,” *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, Nov., 1999.

[10] J. T. Kuehn and B. J. Smith, “The horizon supercomputing system: architecture and software”, *Proceeding of Supercomputing '88*, Nov. 1988., p.p. 28 – 34.

[11] H. Gad, K. Moshe, M. Bilha and E. Vadim, “Light Weight Optimization for Reducing Hot Saves and Restores of Callee-Saved Registers", *4th workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Austin, Texas, Dec. 1, 2001.

[12] Kane, Gerry, *MIPS RISC architecture*, Prentice Hall, 1988.

[13] Rudberg, M.K.; Wanhammar, L., “High speed pipelined parallel Huffman decoding,” *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, vol. 3, 1997, p.p. 2080 - 2083.

[14] Saumya Debray, William Evans, and Robert Muth, “Compiler techniques for code compression,” *Workshop on Compiler Support for System Software*, May 1999.

Code Cache Management Schemes for Dynamic Optimizers

Kim Hazelwood Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University
{hazelwood, smith}@eecs.harvard.edu

Abstract

A dynamic optimizer is a software-based system that performs code modifications at runtime, and several such systems have been proposed over the past several years. These systems typically perform optimization on the level of an instruction trace, and most use caching mechanisms to store recently optimized portions of code. Since the dynamic optimizers produce variable-length code traces that are modified copies of portions of the original executable, a code cache management scheme must deal with the difficult problem of caching objects that vary in size and cannot be subdivided without adding extra jump instructions. Because of these constraints, many dynamic optimizers have chosen unsophisticated schemes, such as flushing the entire cache when it becomes full. Flushing minimizes the overhead of cache management but tends to discard many useful traces. This paper evaluates several alternative cache management schemes that identify and remove only enough traces to make room for a new trace. We find that by treating the code cache as a circular buffer, we can reduce the code cache miss rate by half of that achieved by flushing. Furthermore, this approach adds very little bookkeeping overhead and avoids the problems associated with code cache fragmentation. These characteristics are extremely important in a dynamic system since more complex strategies will do more harm than good if the overhead is too high.

1. Introduction

Dynamic optimization encompasses the idea of applying code optimizations to existing program binaries at runtime. The benefits range from leveraging runtime information to supporting technology for commercial approaches, such as Java. A dynamic optimizer works by observing runtime user behavior and runtime constants, then using that data as a guide for performing optimizations on frequently-executed segments of code. These optimizations may include code re-layout, function inlining, and constant/copy propagation, among others. Following optimization, the new code segment is stored in a code cache. Execution of the optimized code segments occurs directly from the code cache for the remainder of the current program execution (or until the code segment is flushed from the code cache). Due to the increased instruction locality and code specialization, speedups are often achieved. In fact, recent implementations of dynamic optimization systems have achieved speedup values averaging 7% over +02 optimized code [2]. The major tradeoff of dynamic optimization is that, unlike static optimization passes, the time required to observe runtime behavior, perform optimizations, and update program code directly impacts runtime performance. It is very important to keep the overhead to a minimum, or we may lose the benefits of dynamic optimization altogether.

One method for reducing the overhead of dynamic optimization is to make smarter choices regarding code cache management. Because it is not feasible to maintain all optimized code traces produced during an execution in a single code cache, a cache management scheme must be employed. The management scheme should have low overhead and should exploit temporal locality by attempting to keep useful, active code in the dynamic optimizer's code cache. If the same portions of code are repeatedly flushed and regenerated in the cache, then we clearly need to take a different approach when deciding which portions of code are stale and should be flushed. Yet, complex cache management strategies may do more harm than good if the overhead of the scheme is too high. A middle ground that balances the benefits of smarter management choices with the complexity of a management algorithm should be thoroughly investigated.

Several dynamic optimization systems exist that could benefit from smarter cache management, among them are Dynamo [2], Mojo [5], and Wiggins-Redstone [6]. Each system works by (1) performing runtime profiling to determine *hot traces* (frequently executed portions of contiguous code), (2) copying the hot traces into a software-based cache mechanism (possibly performing optimizations on the traces en route), and (3) executing

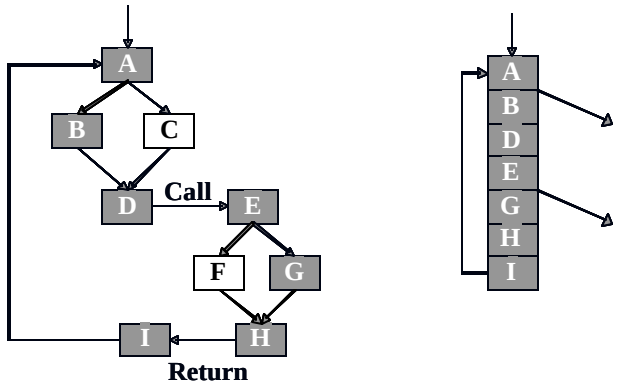


Figure 1 - Example of a code trace

future instances of the hot traces directly from the code cache.

This paper explores cache management strategies as they apply to variably-sized elements and analyzes five alternatives to the currently implemented full cache flush scheme. While the results of this paper were produced with a dynamic optimizer in mind, this work could be equally applicable to dynamic translators and hardware-based code caching mechanisms. The remainder of the paper is organized as follows. Section 2 discusses our experimental methodologies, defines terminology, and presents background data that is used as the foundation of our research. Section 3 discusses the issue of fragmentation that arises when we deal with elements of various sizes. Section 4 discusses several proposed cache management strategies, which are evaluated in terms of resulting miss rate, overhead, and fragmentation in Section 5. Finally, Section 5.2 presents related work and Section 7 concludes.

2. Methodology

Throughout the paper, we use the term *traces*. A trace is a superblock region [9] that is typically used as a basis for optimization (see Figure 1). Traces contain a single entry point and multiple exit points. Internal loops and side entries are not allowed, however a trace may span procedural boundaries. Inside the code cache, a trace is laid out as shown in the right half of Figure 1. Code is often duplicated and specialized within the code cache. For example, a second code trace may include the function call starting with block E in Figure 1, but the procedure may be specialized by choosing the EF-H path instead of the E-G-H path shown in the figure.

To provide a feel for the size of a typical code trace, Table 1 shows the average size of a code trace produced by the Dynamo system running on x86 (as discussed in Section 2.1). Across the SPECint2000 benchmarks using reference inputs, the average trace size is 106 bytes. Yet, as we can see from Figure 2 and the standard deviation column of Table 1, the sizes of individual traces vary greatly during execution of a single benchmark. Table 2 then shows us the number of distinct traces that are produced throughout a single execution of each benchmark. On average, we may experience anywhere from 1,200 to nearly 45,000 traces per execution. Based on the average trace size (from Table 1), Table 3 then shows us that while we can typically fit around 10,000 traces in a 1 MB code cache, we can only fit 625 in a 64 KB cache. This clarifies our point that even for larger code caches, all traces cannot reside in the code cache throughout program execution and emphasizes the need for smart code cache management.

2.1. Our Execution Environment

All results used in this paper were generated using a research version of HP Labs' Dynamo 2.0 dynamic optimizer on an Intel Pentium-II based machine running RedHat Linux 6.2. The Dynamo 2.0 research tool was described by Bruening et al [4] and Smith [11]. Targeted for x86 architectures running Linux or Win32, Dynamo 2.0 differs from that as described by Bala et al [2]. It is now a research tool equipped with several application programming interface (API) hooks to allow information

	avg	min	max	stadev
gzip	97	41	1742	88.27
vpr	102	41	2223	122.23
gcc	96	41	4450	97.09
mcf	96	41	3310	194.75
crafty	121	41	3431	129.60
parser	105	41	1931	92.16
eon	126	41	4810	307.49
perlbmk	90	41	4253	103.80
gap	101	41	4044	105.36
vortex	115	41	4978	138.95
bzip2	99	41	2085	105.24
twolf	126	41	2628	196.33
average	106	41	3324	140.11

Table 1 – Size (in bytes) of code traces produced by Dynamo.

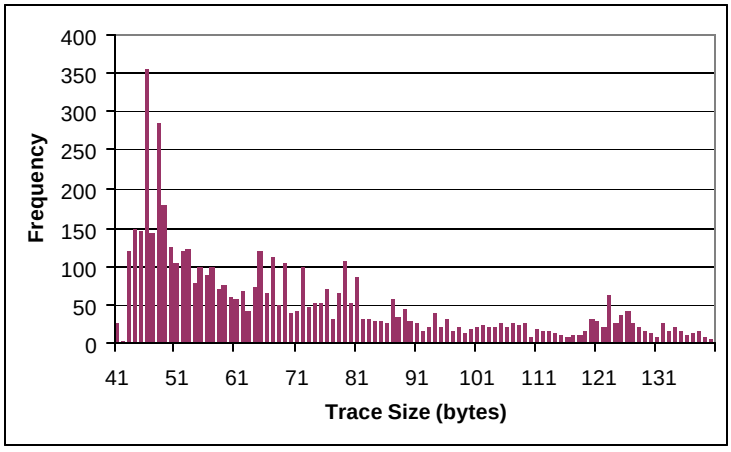


Figure 2 - Distribution of trace sizes for 186.crafty.

gzip	1542
vpr	5055
gcc	44220
mcf	1272
crafty	7153
parser	7856
eon	6238
perlbmk	15959
gap	9827
vortex	13114
bzip2	1624
twolf	6572
average	10036

Table 2 – Distinct number of code traces produced during execution. If we omit gcc as an outlier, the average drops to 6928.

	1 MB	*512 KB*	*256 KB*	*128 KB*	*64 KB*
gzip	10810	5405	2703	1351	676
vpr	10280	5140	2570	1285	643
gcc	10923	5461	2731	1365	683
mcf	10923	5461	2731	1365	683
crafty	8666	4333	2166	1083	542
parser	9986	4993	2497	1248	624
eon	8322	4161	2081	1040	520
perlbmk	11651	5825	2913	1456	728
gap	10382	5191	2595	1298	649
vortex	9118	4559	2280	1140	570
bzip2	10592	5296	2648	1324	662
twolf	8322	4161	2081	1040	520
average	9998	4999	2499	1250	625

Table 3 - Number of traces in a typical code cache for each benchmark and cache size.

regarding the progress of dynamic optimization to be relayed to the user, while the source code and internals remain a black box. The industrial version of Dynamo tracked its own progress and provided an automatic bailout mechanism when it recognized that native execution would be better; however, the research version does not provide this functionality.

We used all twelve SPEC2000 integer programs to generate results. The official test, training, and reference inputs were run to completion under the control of Dynamo using the SPEC2000 *runspec* script. We set environment variables indicating that Dynamo was to dump a trace of all code cache accesses, insertions, and evictions. We then sent this trace through a simulator that implemented the various code cache replacement schemes and analyzed the results.

3. Code Cache Fragmentation

An important issue that must be considered when designing a dynamic optimization cache management scheme is the problem of code cache fragmentation. Just as our hard disk becomes fragmented over time when we create and delete variably-sized files, the cache of the dynamic optimizer may also become fragmented. But because trace generation and replacement occur so frequently in a dynamic optimizer, the problem cannot be ignored.

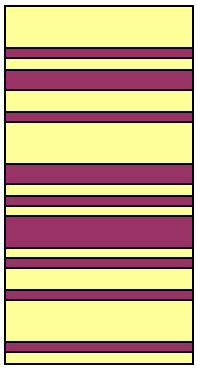


Figure 3 - A Fragmented Code Cache

Figure 3 shows an example of a fragmented cache. The darker areas indicate free space. Consider the case where a fragment of size 1 KB must be inserted into the cache. While the sum of the free space in the cache may add up to 1 KB, a *contiguous* segment of size 1 KB is not available. In this case, we can either employ an expensive defragment operation or simply lose any fragmented free space that is too small to store a code trace. Fragmentation is a serious issue; most known techniques for defragmentation are much too expensive to be implemented in a runtime system. Yet the need for contiguous free space in a code cache necessitates either a low-overhead runtime defragmentation solution, or a management scheme that avoids fragmentation with the code cache altogether.

4. Cache Management Strategies

The problem of cache management in a dynamic optimization system is much more complicated than the standard tasks of instruction and data cache management in modern microprocessors. Unlike hardware caching mechanisms, which focus on replicating data to a location closer to the CPU, dynamic optimizers use caching to create a space in memory where they have the freedom to modify the contents and layout of a dynamic sequence of code. Furthermore, while data and instructions can be cached in fixed-sized blocks, requiring the same partitioning for code traces would negatively interact with the performance optimizations and essentially jeopardize the performance benefits of the optimized segment.

For these reasons, many dynamic optimization systems were designed to perform aggressive, unsophisticated code cache management. The original Dynamo system [2] is one example. As the code cache of the Dynamo system reaches full capacity, the entire cache is flushed in order to make room for new optimized fragments. While attempts are made to recognize changes in an application's working set and preemptively flush the code cache, this is not always possible before the cache fills. Overall, the main

	1 MB	512 KB	256 KB	128 KB	64 KB
164.gzip	5	5	5	5	48
175.vpr	2	2	5	10	139
176.gcc	169	1415	4613	18951	222522
181.mcf	1	1	1	1	3
186.crafty	1	17	2266	10200	154493
197.parser	1	167	3024	6525	18456
252.eon	3	6	936	5277	12952
253.perlbmk	16	197	2283	15656	73125
254.gap	1	4	19	692	9735
255.vortex	6	254	3509	28810	116378
256.bzip2	3	3	3	6	17
300.twolf	1	2	5	104	3879

Table 4 - Number of code cache flushes that occur during execution of the ref input set of SPECint2000 for varying code cache sizes. Because a cache flush always occurs during program exit, the possible number of flushes that could be reported never falls below one.

gzip	84.6%
vpr	38.7%
gcc	42.9%
mcf	29.9%
crafty	58.8%
parser	80.5%
eon	61.5%
perlbmk	63.3%
gap	64.5%
vortex	65.6%
bzip2	81.3%
twolf	66.1%
average	61.5%

Table 5 – Percentage of flushed traces that are later regenerated in the cache, averaged over the five cache sizes.

motivation for flushing is to capture phase changes within a program and leverage these changes for simple cache management.

We can envision several other design motivations behind a code cache management scheme. In particular, the manager can leverage temporal locality of program code, the overall frequency count of program code, or even the size of a code trace when deciding which trace to evict from the code cache. While we expect schemes that focus on temporal locality to perform best, we must also take into account overhead of the schemes, as it could potentially negate the benefits of smarter cache management. In the following subsection, we discuss the current full cache flush scheme, along with five alternative cache management strategies that are based on one of these three motivating factors.

4.1. Full Cache Flush

One cache management scheme that is currently employed is the full code cache flush mechanism. Traces begin filling the cache at its lowest address and continue filling toward higher addresses. As soon as a trace is encountered that cannot be inserted into the cache, all traces are flushed, and the current trace becomes the first element inserted into the empty code cache. While this is a very low-overhead cache management strategy, it has the adverse side effect of flushing hot traces from the cache. If these hot traces are subsequently rebuilt and reinserted into the cache, unnecessary overhead is encountered. Table 4 shows the number of full cache flushes that occur during execution of each of the Spec benchmarks under the control of Dynamo. Table 4 shows us that as the code cache size decreases linearly, the number of cache flushes increases exponentially. For embedded or other memory-restricted systems, this large number of code cache flushes will certainly limit the performance benefits attainable by the dynamic optimizer. Yet, even standard systems cannot be expected to keep pace with trends in software code sizes. Furthermore, Table 5 shows us that 61.5% of the flushed traces are regenerated in the cache, clarifying our point that useful traces are often flushed from the cache and

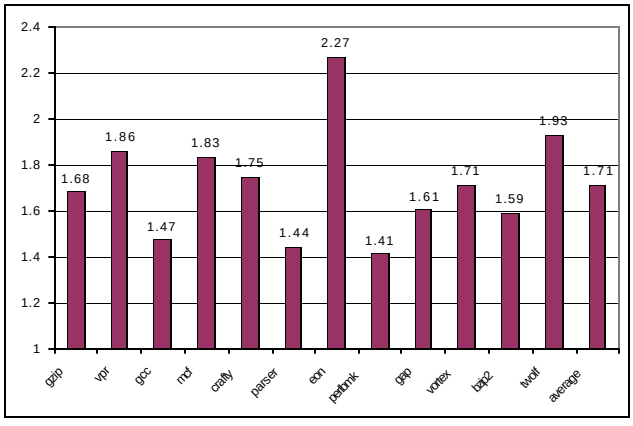


Figure 4 - Average number of trace evictions under LRA before a new element can be inserted.

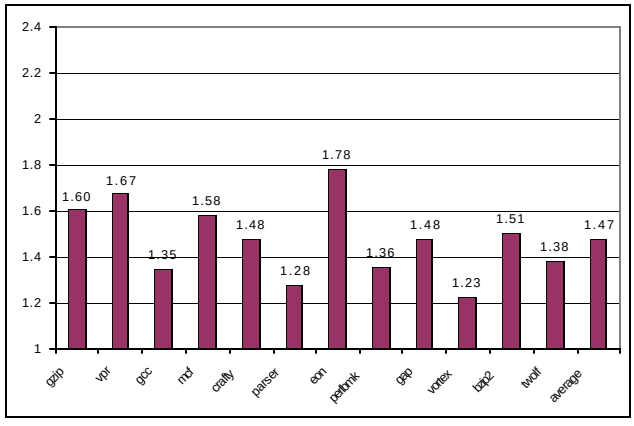


Figure 5 - Average number of trace evictions under LFA before a new element can be inserted.

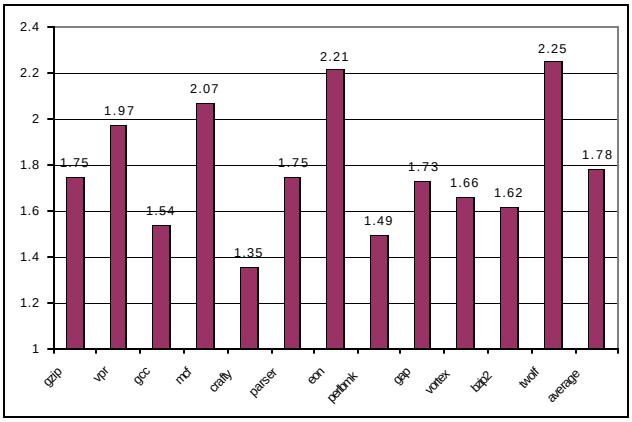


Figure 6 - Average number of fragments removed per insertion using the LRC removal scheme.

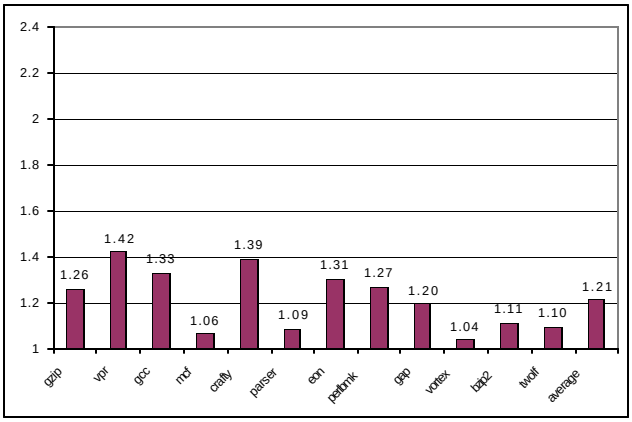


Figure 7 - Average number of fragments removed per insertion using the LE removal scheme.

unnecessary overhead is encountered to regenerate those traces.

4.2. Least-Recently Accessed

An alternative cache management strategy is the Least-Recently Accessed (LRA) strategy. LRA attempts to recognize phase changes in the code by removing traces that have not been accessed recently. For this scheme, we again insert traces into the cache in the order that they are created. However, when the code cache becomes full, the trace that has not been accessed in the greatest amount of time becomes the first candidate for eviction. In the event that this trace will not free enough space to hold the newly optimized trace, the subsequent trace(s) in the code cache are also evicted (since we need to free enough *contiguous* space for the new trace). This scheme has the benefit of leveraging temporal locality, yet it has the unfortunate side-effect of removing innocent victim traces from the cache in order to create adequate contiguous space. Figure 4 gives an indication of the average number of evictions that typically occur in order to make room for the insertion of a single trace. From this figure, we see that it is usually necessary to evict one to two traces from the cache for each new trace that is inserted (or 70% of the time, an eviction will result in the eviction of one additional victim trace.) A second side-effect of this scheme is that it will create code cache fragmentation.

4.3. Least-Frequently Accessed

By maintaining a counter indicating the number of accesses to each optimized trace in the code cache, we can determine the Least-Frequently Accessed (LFA) element. As under LRA, we evict the LFA trace and any subsequent traces necessary to make enough room for the replacement trace. Yet, also like the LRA scheme, LFA will suffer from the effects of code cache fragmentation. Figure 5 shows that 47% of evictions result in a victim trace eviction, thus victims are evicted at a lower rate than LRA. Finally, while this scheme will effectively recognize hot traces and allow them to remain in the code cache, it may have the adverse effect of removing *new* traces from the cache, which haven't yet acquired a high frequency count.

4.4. Least-Recently Created

One of the simplest temporal locality code cache management strategies is a model that treats the code cache as if it were a circular buffer. Called the Least-Recently Created (LRC) method, traces in the code cache are replaced in the same order as they were inserted. In the case where the next candidate for replacement will not free enough space in the code cache for the new trace, subsequent traces are also removed. Though the average number of fragments removed is larger than LRU and LFU, the victim traces removed in this scheme were already next in line for eviction. In the case where an evicted trace frees much more space than is needed for the newly optimized trace, the free space will be used by the next trace inserted into the code cache, thus avoiding fragmentation. In terms of bookkeeping overhead, we merely update a pointer after each trace insertion. And, by proactively removing additional LRC traces from the code cache, a limited amount of additional cache management overhead can be eliminated in the future. Without prior knowledge of the size of the replacement element, the previous schemes, LRA and LFU, could not effectively remove elements in a proactive manner.

4.5. Largest Element

The next two schemes we explored place priority on the size of the element we were attempting to insert into the code cache. The first of these schemes works by evicting the largest trace in the code cache. Called the Largest Element (LE) strategy, it works to minimize the number of evictions that must occur within the code cache, but with no interest placed on temporal locality. Again, subsequent victim traces are removed when the largest element does not produce enough free space for the new trace. Figure 7 (when compared with Figure 4 through Figure 6) shows that the average number of

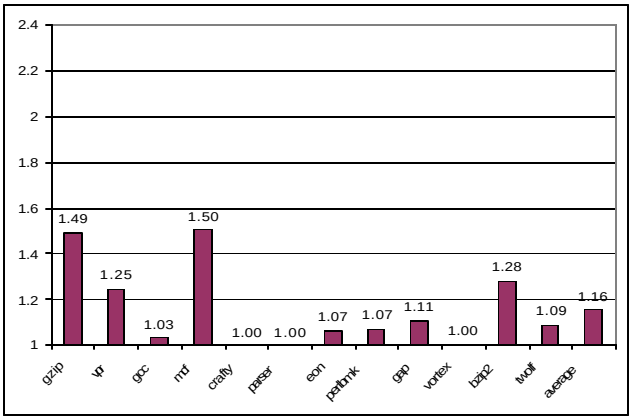


Figure 8 - Average number of elements that must be removed when attempting to find the best-fit element.

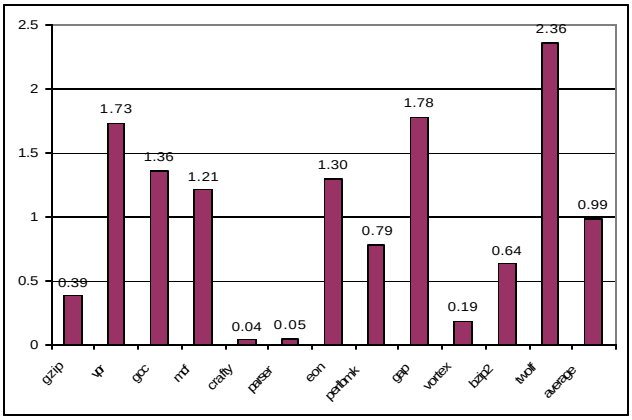


Figure 9 – Average size difference (bytes) between best-fit candidate and element to be inserted into code cache.

evictions per insertion that is necessary using the LE replacement scheme is much lower than the other schemes (dropping from as high as 1.78 down to 1.21). Yet this scheme will suffer from fragmentation, and the bookkeeping overhead includes maintaining a sorted list of the sizes of each trace in the code cache.

4.6. Best-Fit Element

The final strategy we explored was one that attempts to minimize fragmentation by searching the code cache for the best-fit trace to evict. In the Best-Fit Element (BFE) scheme, the code cache is scanned in search of the smallest element that is greater than or equal to the size of the newly optimized trace. When the best-fit trace is found, it is evicted from the cache, ideally leaving just enough room for the new trace. In the case where all traces in the cache are smaller than the new trace, traces are then grouped in subsequent pairs of two, and the best-fit search continues. Figure 8 shows that in 5 out of 6 instances, only one best-fit trace must be evicted from the cache to make room for an incoming trace. In fact, as Figure 9 indicates, we can usually find an eviction candidate that is within one byte of the best-fit size. But this scheme will have the highest overhead of all schemes we have presented, as we may have to do multiple scans of the trace sizes.

5. Results

We simulated the cache management schemes described in Sections 4.1 - 4.6 on a Pentium II-based system with the RedHat Linux 6.2 operating system. Traces were generated by a full-length execution of each of the SPECint2000 benchmarks running under the control of the Dynamo 2.0 dynamic optimization research tool. Because Dynamo was released to us as a black box, we were not able to implement our strategies directly within their system, thus we used the verbose output from Dynamo, which logs the code cache insertions, deletions, and accesses. This log was sent through our own trace-driven simulator, which implemented each code cache replacement policy. A portion of the verbose output generated by Dynamo is shown in Figure 10. Lines 1 and 4 indicate the insertion of a trace into the cache. The traces are numbered and contain a tag indicating their original starting address and trace size in bytes. Lines 2 and 5 indicate an entry into the code cache in order to access a trace. Listed is the original pc as well as the pc within the code cache. A more verbose level of output would also show the exact code within a trace.

5.1. Code cache miss rates

Figure 11 shows the miss rate of each of our code cache management schemes for various code cache sizes, averaged across all benchmarks. From this graph, we see that the best performers are LRC and LRA, regardless of cache size. Both LRC and LRA focus on temporal locality and effectively detect changes in the application's working set. The worst performers, LE and BFE, both focus on trace size rather than temporal locality. And while the graph indicates that the frequency of accesses is more important than trace size (LFA performed better than LE and BFE for small cache sizes), frequency is still not as important as temporal locality. From Figure 11, we can therefore deduce that the most important metric in code cache replacement is temporal locality as expected.

Figure 12 more clearly depicts the effects of cache size on the miss rate of each replacement scheme. Most schemes tend to follow a similar (nearly linear) trend with the exception of BFE and LE. While BFE performs

```
(1)  Fragment 1, tag 0x4013a173, size 45
(2)    Entry into F1(0x4013a173).0x401ac000
(3)    Exit from F1(0x4013a173).0x401ac018
(4)  Fragment 2, tag 0x080488cd, size 61
(5)    Entry into F2(0x80488cd).0x401ac040
(6)    Exit from F2(0x80488cd).0x401ac068
```

Figure 10 – Sample verbose output from Dynamo.

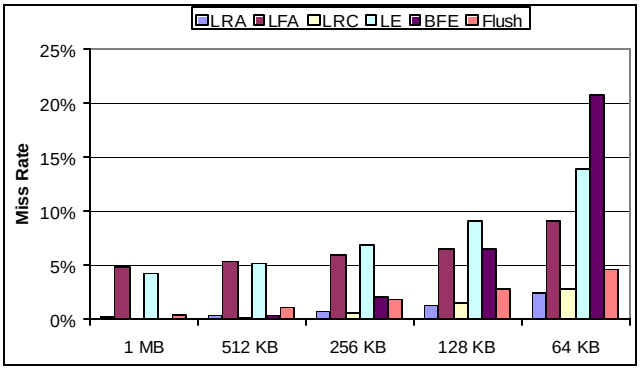


Figure 11 - Miss rate of each code cache replacement scheme for various code cache sizes.

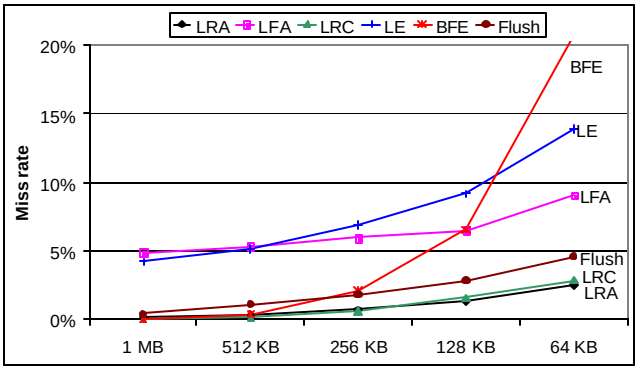


Figure 12 – Effect of cache size on miss rate for various replacement schemes.

well in our large code caches, we see a sharp spike in miss rates as we move to smaller caches. This may be because the hottest traces tend to be similar sizes, and thus the BFE scheme continuously removes useful traces. LE appears to be more affected by decreases in code cache size than most of the other schemes, as its data points appear to rise faster than all others except BFE. This could indicate that the most useful traces in the code cache happen to also be the largest.

While Figure 11 and Figure 12 showed results averaged over all of the benchmarks, Figure 13 takes a look at the performance of the six replacement schemes for each benchmark individually, using a fixed code cache size of 64 KB. We chose 64 KB because nearly all replacement schemes performed well when we dealt with a large code cache, and we felt that investigating the schemes on a small code cache would provide more insight. From the graph, we see several noticeable spikes for the BFE scheme for 186.crafty, 197.parser, and 255.vortex. Yet all other schemes perform very well for these benchmarks. One possible explanation is that for these benchmarks, the typical working set of traces may contain several similarly-sized elements, which continuously replace each other in the code cache using the BFE model. And by revisiting Figure 2, we can see that for crafty, the offending trace size is probably at 46 bytes where we see a spike in the graph. We also notice that while all replacement schemes perform extremely well on 164.gzip (less than 1% miss rate), they all consistently perform poorly on 176.gcc (all over 20% miss rates). For these two cases, the miss rate is clearly dominated by the relatively large or small working set, rather than the replacement scheme. We can verify this by revisiting Table 2, where we notice that 176.gcc produces 44,220 traces during execution, while 164.gzip produces only 1,542 traces.

5.2. Results Summary

As we combine our resulting miss rates with our discussion on fragmentation and complexity of cache management, we can make various conclusions regarding the effectiveness of each code cache replacement scheme. The best performers in the miss rate category were LRA, LRC, and Flush with overall miss rates of 2.48%, 2.88%, and 4.61% respectively (see

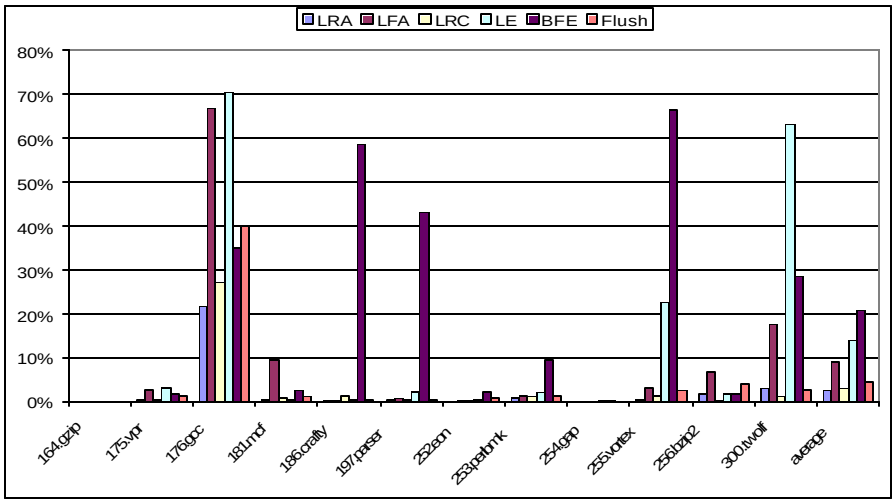


Figure 13 - Code cache miss rates with various replacement schemes and a fixed code cache size of 64KB.

scheme	fragment-ation	additional victims	miss rate	management
LRA	Yes	71%	2.48%	sorted list
LFA	Yes	41%	9.11%	sorted list
LRC	None	78%	2.88%	pointer
LE	Yes	21%	13.91%	sorted list
BFE	Minimal	16%	20.77%	multiple sorted lists
Flush	None	N/A	4.61%	pointer

Table 6 - Summary of each scheme.

Table 6). Yet, the LRA scheme will suffer from the effects of code cache fragmentation, and must be combined with either periodic flushing or code cache defragmentation. In fact, of the cache management strategies explored in this paper, only two did not suffer from the problem of fragmentation. For obvious reasons, full cache flushing avoids fragmentation. In LRC, replacement occurs in a circular manner, thus any free space left over after a replacement will be filled during the subsequent replacements. In terms of overhead, the LRC scheme must simply maintain a pointer to the next free location in the code cache (treating the cache as a circular buffer). The minimal code cache maintenance, combined with a very low miss rate make the LRC scheme a very attractive alternative to the more drastic scheme of full code cache flushing.

6. Related Work

Several groups are currently developing dynamic optimization systems. Dynamo is a system developed at HP Labs that provides a software-based mechanism for selecting and optimizing program fragments [2][4][11]. Wiggins/Redstone is a dynamic optimization and specialization system developed at Compaq [6]. Mojo, developed at Microsoft, is a dynamic optimizer that focuses on x86/WinNT binaries [5]. Dynamo, Wiggins/Redstone, and Mojo all perform optimizations transparently on an unaltered binary at runtime, while storing code traces in a software-based code cache.

There exists a smattering of prior work in the area of improving the performance of dynamic optimizers. Several researchers have proposed lightweight optimizations that are tailored for runtime execution [8][12][14]. Another major interest area has been in techniques to reduce the cost of monitoring application behavior [10][3] and then applying optimizations only to the hottest portions of the executable [1]. There is certainly a large body of work that discusses caching and cache management. As stated earlier, we are restricted in the kinds of cache management approaches we can use because we cache variable length items and must maintain the contiguity of the entire cached element. Under these restrictions, the closest related work also appears in the area of memory overlays before the widespread use of virtual memory [7].

7. Conclusions

Code cache management in the dynamic optimization and translation domains are a crucial, but particularly challenging issue. The high cost of preparing traces for insertion into a cache, combined with the fragmentation issues involved in replacing variably-sized elements has caused many dynamic optimization system developers to sidestep the issue and implement either an enormously large code cache, or an unsophisticated replacement scheme such as full cache flushing. This paper explored five alternatives to the full flush model, and discussed the benefits and tradeoffs of each model. By weighing the factors of (1) code cache miss rate, (2) fragmentation, and (3) complexity of code cache management, we found the Least-Recently Created (LRC) scheme to be a viable solution that succeeds in reducing the code cache miss rate by nearly half of that achieved by the full flush model.

Our future work involves extending our research to include an investigation of the effects of multithreading, interrupts and context switches on code cache management. In addition, we hope to investigate various hybrid or adaptive code cache management schemes. Finally, we hope to directly implement these code cache management schemes in a dynamic optimizer to get more specific details regarding the overhead of each scheme.

Acknowledgments

We would like to thank Hewlett-Packard Laboratories Cambridge for the use of Dynamo. We also wish to acknowledge the independent reviewers for their feedback on an earlier version of this paper, as well as Tom Conte for his insight into an early version of this research. Kim Hazelwood and Michael D. Smith are funded by research grants from Compaq, Hewlett-Packard, IBM, Intel, and Microsoft.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. “Adaptive Optimization in

the Jalapeño JVM," Proceedings of 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00), Minneapolis, Minnesota, October 15-19, 2000.

[2] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System." Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation. 2000, pp. 1-12.

[3] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo." HP Labs Technical Report HPL-1999-78.

[4] Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows." Fourth ACM Workshop on Feedback-Directed and Dynamic Optimization, 2001.

[5] W-K. Chen, S. Lerner, R. Chaiken, D. Gillies. "Mojo: A Dynamic Optimization System." Third ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000, pp. 81-90.

[6] D. Deaver, R. Gorton and N. Rubin. "Wiggins/Redstone: An On-line Program Specializer." Proceedings of IEEE Hot Chips XI Conference, August 1999.

[7] Peter J. Denning. "Before Memory was Virtual." From the book *In the Beginning: Recollections of Software Pioneeers*, IEEE Press, 1997.

[8] Kim M. Hazelwood and Thomas M. Conte. "A Lightweight Algorithm for Dynamic If-Conversion during Dynamic Optimization," Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00) Philadelphia, PA. October 2000, pp. 71-80.

[9] W. Hwu, et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*. Boston: Kluwer Academic Publishers, May 1993.

[10] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu. "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," Proceedings of the 26th International Symposium on Computer Architecture, May, 1999, pp. 136-147.

[11] Michael D. Smith, "Dynamic Optimization: An Online Opportunity." Keynote Speech. 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00) Philadelphia, PA. October 2000.

[12] Michael D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," Proceedings of the ACM Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00) Boston, MA. January 2000.

[13] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. http://www.spec.org/osg/cpu2000/.

[14] Omri Traub, Glenn Holloway, and Michael D. Smith. "Quality and Speed in Linear-Scan Register Allocation," Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 142-151, June 1998.