The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel® Itanium™ Processor

Youngsoo Choi, Allan Knies, Luke Gerke, Tin-Fook Ngai
Intel Corporation, MS SC12-304
2200 Mission College Blvd
Santa Clara, CA 95052
{youngsoo.choi, allan.knies, luke.c.gerke, tin-fook.ngai}@intel.com

Abstract

The research community has studied if-conversion for many years. However, due to the lack of existing hardware, studies were conducted by simulating code generated by experimental compilers. This paper presents the first comprehensive study of the use of predication to implement if-conversion on production hardware with a near-production compiler. To better understand trends in the measurements, we generated binaries at three increasing levels of if-conversion aggressiveness. For each level, we gathered data regarding the global runtime effects of if-conversion on overall execution time, register pressure, code size, and branch behavior. Furthermore, we studied the inherent characteristics of program control-flow structure related to branching to help determine fundamental limits of if-conversion. Our results show that on the Itanium™ processor if-conversion could potentially remove 29% of the branch mispredictions in SPEC2000CINT but that this accounts for a substantially smaller overall program speedup than previously reported.

1. Introduction

Although the research community has investigated predicated execution for many years, due to the lack of real hardware to validate their research results, studies were confined to using processor simulators running binaries generated by experimental compilers (often running without any operating system layer). The recent introduction of the Itanium™ processor and associated production software environment provides the first opportunity to perform new analysis of if-conversion under production constraints.

This paper presents the first comprehensive study of the use of predication to implement if-conversion on general-purpose production hardware and software. We examine three basic effects of if-conversion: the impact of if-conversion on code size, register pressure, and branch behavior on the Itanium processor, limitations of our compiler in performing if-conversion, and how inherent program control-flow characteristics affect if-conversion.

Although this study is limited only to impact of if-conversion, predication has a number of other possible uses. For example, external studies [15] and our own internal work have shown that floating-point (FP) codes improve by about 35% when software pipelining is used. Other studies have shown that code layout has substantial impact on instruction cache (Icache) optimization, but none have directly addressed how predication might be used to improve instruction stream behavior beyond that provided by branch removal. Our results indicate that there is potential for optimization of the instruction stream by using predication, although we do not have sufficient data to draw well-founded conclusions in this area. While control and data speculation in the Itanium instruction set provide great freedom to compilers, non-speculative operations such as stores, checks, and some condition computations are difficult to move. In each of these cases, predication can contribute to the overall performance of an application. While all of these techniques are promising, in this paper, we concentrate only on the use of if-conversion and its impact on removing branches and their mispredictions.

Rather than focusing only on compile-time static measurements or only on run-time dynamic measurements, we have developed a methodology that allows us to link consequences of decisions made at compilation time with execution behavior. Runtime behavior was measured by using the Itanium processor's performance monitors, while compile-time information was stored as annotations in the program executable.

In Section 2, we give a brief summary of prior work in the area of predication and related compiler technology. Section 3 describes the experimental setup used in our study. Experimental results are presented in two sections: Section 4 gives our program measurements across benchmarks and compiler settings, and Section 5 shows our analysis of how the inherent control-flow properties of the benchmarks impact the effectiveness of if-conversion.

2. Comparison to Related Work

Predicated execution has been extensively studied [4, 5, 6, 7, 8, 9, 10, 11, 13, 20]. In architectures that support predicated execution, instructions can be tagged with a guarding predicate. Instructions whose predicate is *true* execute as though they are not predicated; instructions whose predicate is *false* execute as NOP instructions. The process of converting sequences of instructions with conditional branches into predicated operations is known as if-conversion [22].

2.1 Architectural Support for Predicated Execution

Architectural support of predicated execution can be traced back to vector mask operations in vector machines [21], the select instruction in Multiflow Trace [19], and restricted predicated instructions in early superscalar architectures such as the conditional move instruction in DEC/Compaq Alpha and SUN SPARC V9 and nullifying instructions in HP PA-RISC. These architectures only partially support predication through special instructions or in special cases. The Cydra 5 was the first wide-issue machine that fully supported predicated execution [18]. Its single-bit iteration control registers could be used to predicate any instruction within a loop. Today, the Itanium instruction set architecture is the only general-purpose architecture that fully supports predication [1] with 64 individually addressable predicate registers and a wide variety of predicate-computing instructions.

2.2 Use of Predicates in Software Pipelining

In [15, 16], predication was demonstrated to be an important feature to help support software pipelining (SWP) of loops. In addition to removing control dependencies, stage predicates allow the loop kernel to be filled and drained without the use of explicit prologs and epilogs. The Itanium instruction set architecture also supports rotating registers, further increasing the efficiency of SWP. Warter et. al. showed that software pipelined loops performed 34% faster on average with predication than without [15]. Our own experience has shown similar results.

2.3 Compilation of Predicated Code

To gain performance enhancements from predication, different compilation technologies are required. The Illinois IMPACT group proposed the hyperblock structure for predicated execution [6]. A hyperblock is a single-entry multiple-exit region that gives a larger scheduling region than a basic block or a superblock. After instructions from multiple paths are if-converted into a single block, additional optimizations (such as instruction promotion and merging) can be applied to increase ILP in the predicated block [6, 7].

Havanki, et. al. [14] proposed a non-linear region structure called a treegion that can be used to take advantage of predication.

2.4 Reduction of Branches and Branch Mispredictions via Predicated Execution

One major benefit of if-conversion comes from the elimination of hard-to-predict branches. Malhke et. al. [4] studied the impact of if-conversion on branch prediction. It was shown that an average of 27% of branches and 56% of mispredictions were eliminated at runtime with if-conversion over a mix of SPEC92 benchmarks and UNIX utilities. Branches were categorized according to their types (conditional or unconditional, calls or returns), loop and non-loop related, and locations (in inner loop, outer loop, or straight-line code) and their impact on branch prediction was studied. Our study confirms that if-conversion can provide a very substantial removal in branches and mispredictions on SPEC2000CINT. However, our study also includes a correlation with the fundamental structure of control dependences in addition to branch types.

Tyson [9] analyzed branch elimination using different predication models and their impact on branch prediction schemes. A reduction of up to 30% in the misprediction rate in several branch prediction schemes was reported. In order to identify hard-to-predict branches, Mantripragada and Nicolau [5] proposed using data from branch profiling combined with block sizes and schedule lengths, to selectively perform if-conversion. Their simulations were based on an ISA with partial predicate support on a 4-issue out-of-order processor and showed a reduction of misprediction rate from 1.2% to 55% and cycle speedup from 0.5% to 15.2% on SPECint95 benchmarks.

2.5 Performance of Predicated Execution

Past research studies have shown significant application performance improvement due to predication. Based on emulation-driven simulation, Malhke et.al. [8] reported an average speedup of 63% (35%) of full predicated execution over non-predicated superblock scheduling for a mix of SPEC92 and UNIX utilities on an 8-issue processor that can retire/predict 1(2)-branch(es) per cycle. In the same paper, predication was shown to be able to reduce the number of branches and mispredicted branches by 57% and 38%, respectively.

An average speedup of 30% by predication alone was reported more recently [7] for a slightly different mix of SPEC92, SPEC95 and UNIX utilities

benchmarks on a different machine configuration. The processor used profile-based static branch prediction and had a 6-cycle branch misprediction penalty. Their cycle estimates were taken by profiling execution, multiplying block execution counts by compiler-generated expected cycle counts, and then adding cache and exception effects from a simulator.

Our performance results differ substantially from these earlier works due to a number of differences in experimental conditions. In the remainder of this section, we describe the major factors accounting for the difference in results.

First, the region size used by the compilers in the past studies was improved when if-conversion was allowed. This means that the compiler had a greater scheduling window from which instructions are chosen to schedule when if-conversion was enabled than when it was disabled. In our compiler, instructions are moved and scheduled in regions with arbitrary acyclic control flow. Region size is determined independently from whether if-conversion is enabled or not.

Second, most of the previous studies used equal [7] or greater [8] execution resources than what the Itanium processor provides *except* for the number of branch units and the complexity of their predictors. Thus, those studies had fewer execution resource conflicts (making if-conversion easier), but higher contention for branch units (making if-conversion more valuable). The results [8] showed that going from 1 to 2 branch units substantially reduced the performance gained from predication (almost by half). Since the Itanium processor has 3 branch units, this affect is further magnified.

Third, our results were gathered on a real hardware that has TLB misses, pipeline flushes, cache contention, and OS overhead. In [8], such exceptions were assumed to be deferred when they were not necessary. Since our system cannot always do this, our overhead due to these non-branch related activities is higher, and thus, benefits from better schedules and elimination of mispredicted branches account for a proportionally smaller percent of the total execution time.

Finally, the difference in benchmarks and the subsequent change in execution profile from past studies contributes to a different percent of time spent on branch processing. In SPEC2000CINT on the Itanium processor, we see approximately only 7% of cycles spent servicing branch mispredictions.

3. Experimental Setup and Methodology

3.1 Platform

Our experiments were run on a single C0-step Intel® Itanium™ processor running at 733 Mhz using the 460GX chipset and 1 GB of main memory. The Itanium processor can issue up to six instruction slots per cycle and has two load/store units, four general-purpose ALUs, three branch units (and can perform three branch predictions per clock), and two 82-bit floating-point multiply accumulate units (FMAC). The first level instruction and data caches are 16 KB, 4-way set associative. The second level cache is 96KB unified, 6-way set associative and the third level cache is 4MB, 4-way set associative. A more complete description of the Itanium processor micro-architecture is available in [2].

3.2 Compiler

The executables used in our performance runs were built using a near-production version of Intel's product Itanium architecture compiler, ECC. Enhancements were made that allowed us to adjust the heuristics used when making if-conversion decisions and to insert annotations into the generated binaries. The compiler was otherwise unmodified.

In ECC, predicate generation, along with all predicate-aware optimization, is done in the code generator. If-conversion is performed immediately after software pipelining (when it is enabled) and immediately before global code scheduling. Potential candidates for predication are acyclic single-entry regions, excluding those that contain architecturally unpredicatable instructions or indirect branches. Compile-time and code-size considerations limit our searches to regions with reasonable size. However, once a region is if-converted the resultant block is considered for if-conversion with surrounding blocks, effectively allowing if-conversion of large regions.

Predicate relationships are established through evaluation of the control flow graph and represented as a Predicate Partition Graph [17] for use by downstream phases. The heuristics used to decide whether to if-convert are based on a number of factors including: dependence height, resource height, estimated misprediction rates, and edge profile information.

The compiler performs global code motion using Wavefront Scheduling [12]. The scheduling regions allow arbitrary acyclic control flow, and are thus not directly impacted by the presence or absence of upstream if-conversion. Even in the absence of if-conversion, the compiler assigns predicates to blocks that may be used by the code scheduler to perform

upward and downward code motion. The register allocator removes unused predicates later.

The register allocator is currently only partially predicate-aware. In particular, the register lifetime of a virtual register used inside a loop and defined under a predicate is conservatively thought to extend upward beyond its predicated definition.

3.3 Annotations

Our compiler has been modified to emit compile-time information about specific functions, basic blocks, and instructions into an annotation segment of the binary [3]. These annotations are kept in a non-loadable segment of the binary and have no impact on the actual executable code generated by the compiler. Post-processing tools were developed to read the annotations from the binary and associate the information with dynamic event data collected by the Itanium processor's performance monitors at runtime.

The compiler used this annotation section to record program information related to heuristic decisions made by the if-converter as well as to tag branch types. Each branch is tagged with a type such as loop-back, early loop-exit, call-type, IP-relative, indirect, and controlling single/multiple-entry/exit acyclic regions. These per-instruction records, used together with per-instruction runtime event information, allow us to relate low-level runtime event counts to higher-level program and branch characteristics. In Section 5, this technique is used to characterize branches according to their control-flow properties and their architectural branch types.

3.4 Event sampling

Each binary was run multiple times to record information from hardware performance counters. These counters track a wide variety of basic events, such as instructions retired, cycles, and branches retired. The monitors are highly configurable (see [1], Volume 4) to allow the user to collect event information based on instruction opcode, memory reference address, process privilege level, and other criteria. The Itanium processor also supports event-based sampling through the use of event address registers (EARs).

When using EARs, it is possible to specify the number of times an event occurs before a sample is taken (sampling ratio). When the specified number of events has occurred, the IP of the instruction causing that event is recorded as well as event-specific data, the performance monitors are frozen, and an external interrupt is raised. After the interrupt handler records all relevant information, monitors are unfrozen, and execution continues. In this study, we collected branch event samples and the sampling thresholds were adjusted so that each benchmark had approximately 4 million samples. The sampling does not significantly impact program execution as the overhead was generally less than 3%.

3.5 Benchmark Runs

All experiments were run using the SPEC2000CINT benchmarks under compiler options compliant with SPEC "BASE" setting rules. All runs were performed with the reference input set. Our base settings included profile-guided optimization using execution traces from the training set, whole program inter-procedural optimization, and function inlining. Three versions of each integer benchmark were built to provide three degrees of if-conversion aggressiveness: No if-conversion (NONE), default level of if-conversion (DEF), and complete if-conversion wherever possible (MAX). This classification is based on the if-conversion heuristics described in Section 3.2. The NONE level disallows if-conversion entirely, the DEF level allows the compiler to if-convert where it seems profitable according to heuristics, and the MAX level if-converts every region that the region-constructor creates, regardless of profitability. (Note that due to the compilation resource requirements at the MAX level, we were unable to generate a functional MAX binary for gcc, so the result is missing in our graphs.)

The DEF level has been tuned to maximize the average performance of applications, other levels were expected (and indeed) generally resulted in slower binaries regardless of the amount of if-conversion performed. The MAX level binaries were produced to help us understand how aggressive if-conversion changes control flow structure and associated program characteristics, rather than to improve performance.

As mentioned in Section 2.2, predicates can be used effectively in software pipelining (SWP). In order to focus on the use of predication in if-conversion, we turned off SWP in the compiler so that measurements taken on predicated code only include affects from if-conversion, rather than from SWP stage predicates. Although SWP is turned off in our experiments, the compiler still if-converts code within loops. In SPEC2000CINT, the time spent in loops that can be software pipelined is very small and thus the performance benefit for using this feature is less than 1% on SPEC2000CINT (although for SPEC200FP, the benefit is greater than 30%).

4. Experimental Results – Basic Data

In this section, we present data and analysis regarding if-conversion and its impact on general program behavior using binaries compiled at NONE, DEF, and MAX aggressiveness levels. The following subsections describe the basic program runtime

behavior based on static and dynamic measures and discuss the corresponding microarchitecture and compiler issues.

4.1 Total Execution Time

Benchmark	NONE binary (No if-conversion)				
	CPU Cycle count (million)	% Cycle spent servicing branch misprediction	Instructions retired (million)	NOPs retired (million)	Effective IPC
164.gzip	386,814	9.37%	540,344	162,999	0.98
175.vpr	381,289	6.66%	295,576	95,235	0.53
176.gcc	237,116	7.52%	241,071	56,858	0.78
181.mcf	493,147	1.34%	105,825	29,010	0.16
186.crafty	206,538	9.82%	224,410	46,155	0.86
197.parser	489,667	8.52%	495,401	136,084	0.73
252.eon	242,004	6.46%	316,048	99,284	0.90
253.perlbmk	378,016	8.21%	572,988	150,717	1.12
254.gap	293,162	3.93%	335,921	83,601	0.86
255.vortex	266,340	1.16%	365,243	80,857	1.07
256.bzip2	341,874	8.98%	388,623	87,176	0.88
300.twolf	622,937	11.28%	498,668	166,575	0.53
AVERAGE	361,575	7.15%	365,010	99,546	0.78

Table 1. Basic statistics for binaries with no if-conversion (NONE)

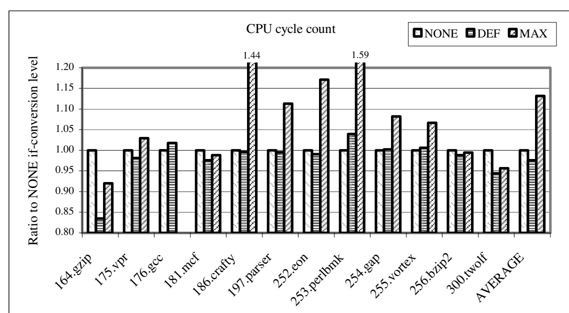

Figure 1. Relative CPU cycle count (NONE = 1.0)

Table 1 shows the total number of execution cycles for the NONE binaries and the estimated percent of time spent in servicing branch mispredictions. Since the cycle numbers reported are measured on near-production silicon with near-production compiler technology, performance will be somewhat lower than what one would expect on a system available by the time this paper is published.

Figure 1 shows the CPU cycle counts of other if-conversion levels relative to those of the NONE binaries given in Table 1. Although we report overall performance at different levels of if-conversion aggressiveness, the focus of our study is to understand the fundamental issues that arise from if-conversion rather than whether a given compilation parameter increases or decreases performance.

Figure 1 shows that more if-conversion does not necessarily lead to increased performance. As expected, the DEF if-conversion level usually results in the highest performance, although there are some variations between benchmarks. On average, the DEF if-conversion level was 2% faster than the NONE binaries, while the MAX if-conversion level was 14% slower than NONE. The remainder of this section discusses related effects and analyzes whether these effects can be improved by changes in microarchitecture or compiler technology.

4.2 Instruction Access Behavior

Because code size and code layout can have subtle effects on program performance, we use three metrics to characterize the effect of if-conversion on the instruction stream: dynamic count of retired instructions (including nops and predicated-off instructions), static instruction count (number of instructions in the binary including nops), and the number of first level instruction (L1I) cache misses.

The number of instructions retired at runtime indicates the minimum bandwidth required between the L1I cache and the execution units. Figure 2 shows that the number of instructions retired increases with more aggressive if-conversion. This is not surprising because the more aggressively if-conversion is applied, the more instructions from secondary control paths are sent down the execution pipe (and later squashed).

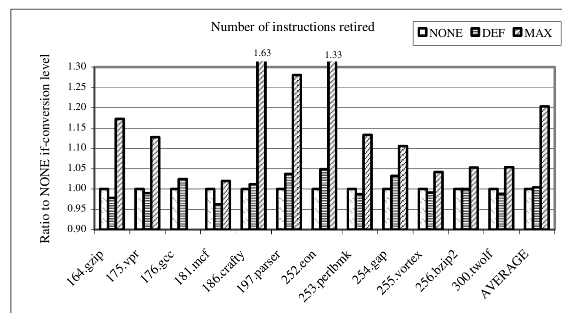

Figure 2. Relative number of instructions retired (NONE = 1.0)

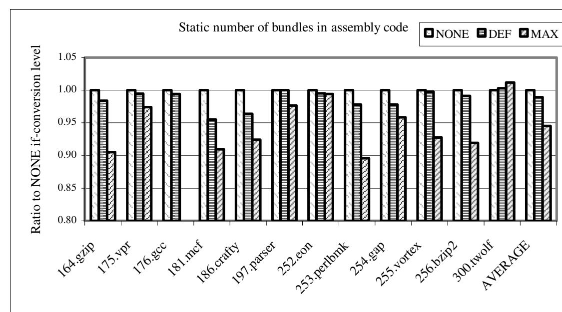

Figure 3. Relative static bundle count changes (NONE=1.0)

Figure 3 shows the static instruction counts for each binary. The static instruction count is an

important measure because it contributes to the load time of the binary, the amount of disk space required for its storage, and is generally an indirect contributor to instruction TLB (ITLB) misses. The code size of DEF and MAX binaries are 1% and 5% smaller, respectively, than that of NONE binaries. The static code size always decreases with increasing predication due to reduction in the number of branches and nops. Nops are decreased because fewer branches, labels, and higher ILP provides the compiler with greater freedom in bundling instructions.

The last measure of the impact of code size is the number of first level instruction (L1I) cache misses incurred during execution. (Figure 4) For gcc, parser, gap, and vortex, increasing if-conversion aggressiveness increased the number of Icache misses. For gzip, mcf, perlbmk, bzip2, and twolf, DEF if-conversion level caused the fewest L1I cache misses (more than 10% fewer in some cases).

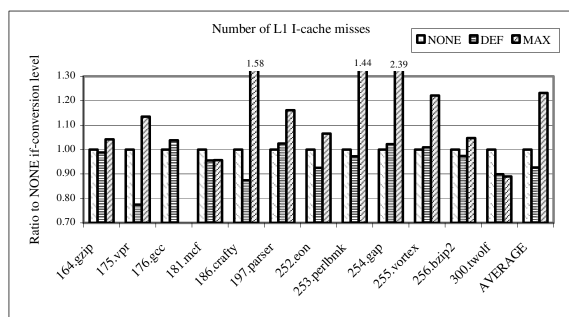

Figure 4. Relative number of L1I cache misses (NONE = 1.0)

In summary, on average, the MAX level produces the smallest static code size while the NONE level retires the least number of instructions (as there are no extra predicated-off instructions). The DEF level incurs the fewest L1I cache misses with slight increases in dynamic instruction count. This indicates that by combining multiple control paths following a branch, the default level if-conversion results in more effective I-cache prefetching while not significantly increasing the footprint of the original code.

4.3 Branch Behavior

Figure 5 shows the total number of branches executed for each benchmark at different levels of if-conversion aggressiveness. In some benchmarks, as if-conversion level increases, successful branch elimination by if-conversion reduces the number of branches retired. However, for gzip, gcc, crafty, parser, and perlbmk, the number of branches retired increases. This can be attributed to the fact that unconditional calls, indirect branches, and returns in secondary control paths, which are not executed at the NONE level, are being predicated and pulled into the program's execution path via if-conversion. Note that, although the MAX level shows 19% more branches on average, the number is skewed by perlbmk and crafty.

Figure 5. Relative number of branch instructions retired (NONE = 1.0)

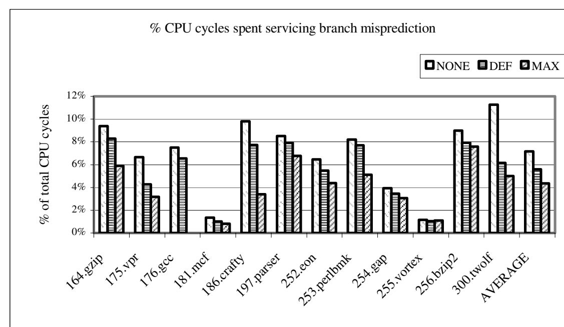

Figure 6. Percentage of CPU cycles spent servicing branch misprediction

Figure 6 shows the percentage of cycles spent servicing branch mispredictions computed by multiplying the number of branch mispredictions by the misprediction penalty on the Itanium processor (10 cycles). We see that there is a substantial reduction in percent of CPU cycles due to branch mispredictions for all benchmarks except perlbmk. In Section 5, we will explain perlbmk's behavior. On average, the DEF level if-conversion reduces branch misprediction cycles by 20% and MAX by 27%. However, as expected, overall execution time spent in servicing mispredicted branches decreases with increasing aggressiveness, varying from 0.2 % to 5% at DEF and from 0.2% to 7% at MAX.

4.4 Compiler Effects

Currently, ECC's register allocator is not fully aware of predicate relationships. While predication should not generally increase the use of general-purpose registers, in some cases, ECC allocates more registers than necessary.

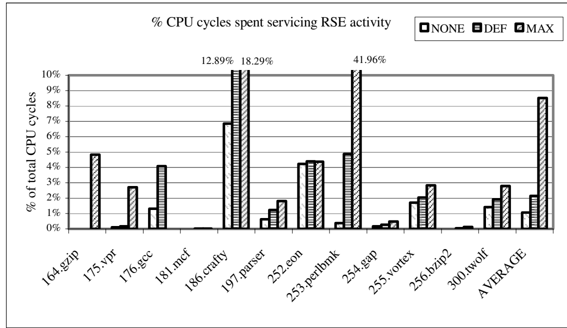

Figure 7. Percentage of CPU cycles spent in servicing RSE activity versus if-conversion level

Figure 7 shows the estimated percent of execution time spent servicing register stack engine (RSE) requests (spilling/filling registers during function call/return sequences)[1], which is a good indication of how well the compiler, did at register allocation. As shown in Figure 7, the percent of time spent in servicing RSE traffic is quite substantial at higher levels of if-conversion in some benchmarks. Once this limitation in ECC is fixed, the absolute performance gain with if-conversion should improve proportionally.

5. Experimental Results – Inherent Control Flow Characteristics

While the results in Section 4 concentrate on the changes in basic program characteristics at varying levels of if-conversion, this section focuses on the inherent properties and control flow structures of the benchmarks. Through a combination of use of compiler annotations, performance hardware feedback, and post-processing tools, we have characterized the inherent control flow structures of the programs by the number, type, distribution, predictability, and context of branches in the source programs. We use this data to establish an upper bound on the improvements that if-conversion can achieve by reducing branch-mispredictions.

5.1 Branch Type Analysis

In this subsection, we examine the type and control-flow properties of branches, identifying those that are not removable by if-conversion. Table 2 summarizes the distribution of branches and mispredictions in NONE binaries according to their branch types. Dynamic data are based on runtime sampled event data and are averaged across the benchmarks. Individual branches are classified either as one of eleven different types of branches that *cannot* be removed via standard application of if-conversion or as belonging to the "Other IP-relative" category.

Branch type (NONE if-conversion binary)		% of total number of branches: static	% of total number of branches: dynamic (sampling)	misprediction rate of each branch type	% of total mispredictions due to the branch type	average % of CPU cycles spent servicing branch misprediction
Nop.b		2.66%				
Call	Indirect call	0.64%	0.62%	43.19%	3.80%	0.28%
Call	IP-relative call	23.47%	3.42%	0.25%	0.12%	0.01%
Return		6.68%	3.91%	9.65%	5.39%	0.31%
Loops	Counted (br.cloop, br.ctop, br,cexit)	0.92%	6.39%	9.78%	8.92%	0.74%
Loops	IP-relative loop back	2.94%	15.86%	6.06%	13.72%	1.01%
Loops	IP-relative loop exit	2.58%	8.01%	6.22%	7.11%	0.43%
Indirect	Indirect unconditional	0.12%	0.34%	41.35%	2.01%	0.21%
Indirect	Indirect conditional	0.00%	0.00%		0.00%	0.00%
IP-relative conditional recovery		1.53%	0.00%	0.00%	0.00%	0.00%
IP-relative unconditional		25.11%	6.63%	2.59%	2.45%	0.17%
Subtotal for **classified branches**		66.64%	45.18%		43.53%	3.15%
Other IP-relative		33.28%	54.81%	7.22%	56.47%	3.77%
Sum		100%	100%		100%	6.92%

Table 2. Average branch behavior by type for NONE binaries

Table 2 shows that eleven types of irremovable branches account for 67% of the static branches, 45% of the dynamic branches, and 44% of the program's mispredictions.

In theory, it is possible to remove some of the mispredictions associated with branches that we have classified as irremovable. However, in practice it is either difficult or not profitable to do so. For example, the conditional calls that remain (the compiler has already performed interprocedural inlining where we thought profitable) could be completely removed via inlining and predication. However, unless the called subroutine itself contains no difficult-to-predicate instructions (such as calls, indirect branches, loops, or unpredicatable instructions), it may not be possible to effectively remove the mispredictions associated with the call. Thus, our definition of irremovable branches generally only refers to the mispredictions caused by branches, rather than whether the branch itself is theoretically removable.

In Table 2, the "Other IP-relative" row accounts for all the branches that do not fall into one of those irremovable categories – these are the branches that form the set of branches that *might* be removable via if-conversion. These branches account for 56% of all mispredictions, and provide a bound on the number of mispredictions that can be removed via if-conversion,

[1] Based on anecdotal evidence gathered from kernels and knowledge of the Itanium processor microarchitecture, each RSE load/store pair is estimated to take 2 cycles.

with some caveats. First, unconditional IP-relative branches are removed as a side-effect of if-conversion or block ordering. Since they cause very few of the mispredictions (2.5%), their removal is not generally going to have large impact on branch prediction. Second, our results are based on the Itanium processor branch predictor, and different branch predictors may give different results.

Benchmarks	% MP due to "Other IP-relative" branch	% MP due to counted loop branch	% MP due to non-counted loop-related branch	% MP due to all indirect branch
164.gzip	76.5%	1.8%	21.0%	0.0%
175.vpr	74.6%	11.6%	12.9%	0.0%
176.gcc	50.0%	9.4%	19.4%	10.0%
181.mcf	49.6%	0.0%	48.9%	0.0%
186.crafty	61.2%	3.0%	17.9%	4.4%
197.parser	51.7%	4.7%	32.9%	0.0%
252.eon	62.9%	0.0%	7.7%	12.7%
253.perlbmk	15.6%	13.5%	9.7%	54.7%
254.gap	34.9%	26.2%	17.5%	13.6%
255.vortex	66.9%	0.2%	8.3%	4.8%
256.bzip2	52.3%	11.6%	29.2%	0.0%
300.twolf	54.4%	20.7%	20.4%	0.0%

Table 3. Misprediction statistics for selected branch types

Table 3 shows that the misprediction statistics vary greatly by benchmark. For example, while the number of mispredictions due to “Other IP-relative” branches is 76% in gzip, it is only 15% for perlbmk. A close look shows that a very large percentage of mispredictions in perlbmk (55%) are due to branch target mispredictions from indirect branches, which are not directly removable via predication (our compiler already peels out common cases from switches, but does not directly attempt to peel mispredicting cases).

Surprisingly, even though the Itanium processor has a dedicated predictor for counted loops, a large percentage of mispredictions still occur. This is most likely due to the fact that the processor must fetch 6 or 7 cache lines of instructions between when the loop count register is set and when the loop terminates in order for the predictor to be accurate. Except in the cases of eon and vortex, loop-related branches account for 20% to 50% of all mispredictions.

This shows that substantial portions of the mispredictions in some benchmarks cannot be removed by if-conversion due to the fundamental nature of the code. In the next section, we will refine the bound on the set of mispredictions that are removable via if-conversion.

5.2 Control Based Classification

In this subsection, we sub-classify “Other IP-relative conditional branches” by the instructions that are control dependent on them. We are interested in five specific types of instructions: calls, returns, loops, indirect non-call branches, and architecturally unpredicatable instructions (alloc, flushrs, loadrs, rfi, bsw, clrrrb, cover, epc). We say a branch *controls* one of these types of instructions when no acyclic path from the branch to the instruction in question passes through any instruction that post-dominates the branch (this is effectively a definition of control-dependence). Furthermore, such control is said to be *direct control* when no acyclic path passes through another conditional branch before reaching the instruction in question. *Indirect control* is a branch with control that is not direct control.

Branches that directly control unpredicatable instructions cannot be removed without altering program behavior. Branches that control call, indirect, and return branches can be removed via if-conversion, but the controlled call, indirect branch, or return will then simply inherit the predicate from the removed branch. If this happens, directional mispredictions from the parent will almost certainly transfer to the child call, indirect, or return branch. If multiple calls or indirect branches are controlled, it is possible that predication will even increase the number of mispredictions by effectively replicating the mispredicting condition on several branches.

Benchmark	% of instructions by calls and indirect branches			% of misprediction due to call and indirect branches		
	NoPrd Binary	BASE Binary	AggPrd Binary	NoPrd Binary	BASE Binary	AggPrd Binary
164.gzip	1.33%	1.32%	2.94%	0.52%	4.17%	12.10%
175.vpr	1.32%	1.53%	4.87%	0.00%	0.00%	0.02%
176.gcc	3.93%	4.45%		10.19%	14.47%	
181.mcf	0.15%	0.30%	0.30%	0.84%	2.67%	3.14%
186.crafty	3.43%	4.95%	9.62%	4.38%	8.72%	11.51%
197.parser	5.24%	6.54%	13.39%	0.12%	1.99%	3.83%
252.eon	10.45%	16.07%	24.73%	12.72%	26.79%	32.84%
253.perlbmk	10.91%	11.61%	25.14%	54.78%	57.98%	66.27%
254.gap	5.97%	8.13%	12.39%	13.68%	17.30%	24.48%
255.vortex	4.67%	5.41%	11.69%	4.79%	9.27%	27.11%
256.bzip2	2.62%	2.65%	14.12%	0.00%	0.00%	0.27%
300.twolf	1.05%	1.49%	2.27%	0.00%	0.00%	0.43%

Table 4. Percentage of calls and indirect branches executed and their mispredictions

Likewise, the mispredictions associated with a branch that directly controls a loop (the loop branch and its body) cannot practically be removed via if-conversion because the loop back branch would then become part of the predicated region and mispredictions could migrate to it from the parent. In Table 4, we can see misprediction migration as the if-conversion level increases by noticing that the percentage of calls and indirect branches executed generally increases.

Similar discussion applies to branches with indirect control, but is more difficult to draw firm conclusions. While it is possible to remove some branches with indirect control, it is possible that all or some of the mispredictions associated with the original parent will migrate to the controlled branches. Thus,

branches with indirect control sometimes can be if-converted, but the resulting misprediction behavior of the formerly controlled branches is difficult to analyze statically.

Control Info of "Other IP-relative" branches	% of total number of branches		misprediction rate	% of total mispredicts	average % of CPU cycles spent servicing mispredictions
	static	dynamic (sampling)			
Some direct control	15.36%	17.18%	5.73%	14.06%	0.98%
Loop entry	1.78%	2.52%	9.60%	3.46%	0.22%
Loop exit	1.75%	5.29%	8.73%	6.60%	0.55%
Call	10.91%	9.23%	3.21%	4.23%	0.24%
Indirect branch	0.10%	0.36%	2.92%	0.15%	0.01%
Return	3.83%	3.16%	6.67%	3.01%	0.16%
Unpredicatable	0.00%	0.00%	0.00%	0.00%	0.00%
Indirect control only	9.97%	16.98%	5.40%	13.08%	0.88%
Loop entry	4.71%	8.23%	6.66%	7.83%	0.53%
Loop exit	1.29%	2.01%	0.35%	2.75%	0.17%
Call	7.12%	10.60%	4.48%	6.77%	0.46%
Indirect branch	0.37%	0.33%	16.70%	0.78%	0.06%
Return	2.53%	2.37%	7.28%	2.46%	0.15%
Unpredicatable	0.00%	0.00%	0.00%	0.00%	0.00%
Does not control	7.94%	20.66%	9.95%	29.33%	1.92%
Subtotal for "Other IP-relative"	33.28%	54.81%		56.47%	3.77%
Classified branches	66.64%	45.18%		43.53%	3.15%
Sum	100%	100%		100%	6.92%

Table 5. Branch behavior based on its control (direct/indirect)

Benchmarks	Other IP-relative branch (removable branch)			
	no control		indirect control only	
	% total misprediction	% total CPU cycles	% total misprediction	% total CPU cycles
164.gzip	30.09%	2.82%	16.97%	1.59%
175.vpr	49.79%	3.32%	16.86%	1.12%
176.gcc	9.37%	0.70%	23.22%	1.75%
181.mcf	33.78%	0.45%	10.08%	0.13%
186.crafty	33.65%	3.30%	13.11%	1.29%
197.parser	3.22%	0.27%	13.82%	1.18%
252.eon	29.27%	1.89%	8.62%	0.56%
253.perlbmk	2.09%	0.17%	6.20%	0.51%
254.gap	15.58%	0.61%	14.02%	0.55%
255.vortex	28.23%	0.33%	28.66%	0.33%
256.bzip2	13.09%	1.18%	18.09%	1.63%
300.twolf	48.45%	5.46%	5.73%	0.65%
AVERAGE	29.33%	1.92%	13.08%	0.88%

Table 6. Branch misprediction information for potentially removable branches

Table 5[2] breaks down branches based on their control characteristics. "Some direct control" includes any branch that directly or indirectly controls one of the six categories of controlled instructions, "Indirectly control only" includes those branches that have indirect control, but no direct control, and the "Does not control" category includes those branches that have no direct or indirect control over any of the six categories.

"Does not control" branches and their mispredictions are completely removable via if-conversion, although there are a variety of factors such as dependence height, resource height, code size, and predicate register usage that may make predication undesirable in a given situation. On a per-benchmark basis, Table 6 shows that the percent of mispredictions due to "Does not control, Other IP-relative branches" is relatively small and, on average, contributes 28% of mispredictions accounting for 2% of the overall execution cycles. If we included "indirectly-controlled IP-relative branches", this would account for 41% of mispredictions and 3% of overall execution time.

According to Table 6, the CPU cycle reduction possible due to reducing branch mispredictions by applying if-conversion on the Itanium processor is about 2-3%. However, this number does not account for potential reductions in capacity or conflict misses in predictors but does provide an upper bound based on the predictability of specific branches. As pointed out in the related research summary, there are many other possible uses for prediction beyond if-conversion. Thus, it would be a mistake to assume that 2-3% is the maximum potential performance gain for predication as a whole.

6. Conclusions

This study presents the first data analyzing the impact of if-conversion on real hardware and production software. We distinguish between if-conversion and predication by noting that if-conversion is but one of several uses for predication. Our study only addresses if-conversion, although both others' and our own results show predication provides substantial benefit for software pipelining, improvements in instruction stream behavior, and ability to perform more advanced code motion.

The data presented in this study provides a realistic evaluation of how if-conversion affects program execution while helping to establish some fundamental limits on the impact of if-conversion. We have also presented data showing how code size, register pressure, and branch behavior are impacted by if-conversion in greater details and in more realistic conditions than has been previously possible.

Although this work and previous research have demonstrated that if-conversion is very effective at removing branches and mispredictions, our results show that differences in compiler technology, hardware resources, and benchmark behavior substantially affect the conclusions of prior estimates of if-conversion potential. However, in the future, further changes in workload or microarchitecture could substantially change the importance of the findings in this paper again. Since follow-on implementations of the Itanium processor are likely to have bigger caches and the ILP compilation techniques that exploit other Itanium features are

[2] The sub-categories do not sum to the row totals because they overlap with each other (one branch could control both a return and a call).

becoming more mature, the portion of execution time dominated by branch penalty will likely grow. In fact, this effect partially accounts for why prior academic studies saw a larger improvement from if-conversion – they assumed that non-branch related factors were smaller than what we have seen on the Itanium processor for SPEC2000CINT.

Finally, we note that predication has major impact on applications that have substantial time in software pipelined loops or that have a very substantial component of time spent in servicing mispredictions. For applications that are Icache-limited, if-conversion combined with profile-feedback directed layout look promising and we expect research to examine this area in depth. Because of these fundamental differences in application codes, we expect to continue to see the impact of predication vary widely across benchmarks and microarchitectures.

7. References

[1] Intel Corporation. Intel® Itanium™ Architecture Software Developer's Manual. Available from http://developer.intel.com/design/ia-64/manuals/, July 2000.

[2] Intel Corporation. Itanium™ Processor Microarchitecture Reference for Software Optimization. Available from http://developer.intel.com/design/ia-64/manuals/, August 2000.

[3] Intel Corporation. Flexible Annotations API Programming Guide. Available from http://developer.intel.com/software/product/opensource/tools/perftools.htm, June 2001.

[4] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen-mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 217–227, 1994.

[5] Srinivas Mantripragada and Alexandru Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. *Proceedings of the 2000 international conference on Supercomputing*, pages 206–214, 2000.

[6] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45-54, 1992.

[7] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran and Wen-mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. *Proceedings of the 25th annual international symposium on Computer architecture*, pages 227-237, 1998.

[8] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. Proceedings of the 22nd annual international symposium on Computer architecture, pages 138-150, 1995.

[9] Gary Scott Tyson. The effects of predicated execution on branch prediction. Proceedings of the 27th annual international symposium on Microarchitecture, pages 196-206, 1994.

[10] Alexandre Eichenberger, Waleed Meleis, and Suman Maradani. An integrated approach to accelerate data and predicate computations in hyperblocks. Proceedings of the 33rd annual IEEE/ACM international symposium on Microarchitecture, pages 101–111, 2000.

[11] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, pages 114–125, 1996.

[12] Jay Bharadwaj, Kishore Menezes, and Chris McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. Proceedings of the 32nd Annual ACM/IEEE international symposium on Microarchitecture, pages 262–271, 1999.

[13] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. Proceedings of the 21ST annual international symposium on Computer architecture, pages 120-129, 1994.

[14] W. A. Havanki, S. Banerjia, T. M. Conte. Treegion scheduling for wide issue processors. Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, pages 266-276, 1998.

[15] N. J. Warter, D. M. Lavery, and W. W. Hwu. The benefit of predicated execution for software pipelining. Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, Vol. I, pages 496-506, 1993.

[16] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. Proceedings of the third international conference on Architectural support for programming languages and operating systems, pages 26-38, 1989.

[17] R. Johnson, M. Schlansker. Analysis Techniques for Predicated Code. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pages 100–113, 1996.

[18] B. R. Rau, D.W.L. Yen, W. Yen, R.A. Towle. The Cydra 5 departmental supercomputer. IEEE Computer, pages 12-35, January 1989.

[19] P.G. Lowney, et. al. The Multiflow trace scheduling compiler. The Journal of Supercomputing, Vol. 7, pages 51-142, January 1993.

[20] P.Y. Chang, E. Hao, Y. Patt. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 1995.

[21] R.M. Russel. The CRAY-1 computer system. CACM, Vol. 21, pages 63-72, January 1978.

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