

# Using Performance Bounds to Guide Pre-scheduling Code Optimizations

Huiyang Zhou

*Department of Computer Science  
University of Central Florida  
[zhou@cs.ucf.edu](mailto:zhou@cs.ucf.edu)*

Thomas M. Conte

*Department of Electrical and Computer Engineering  
North Carolina State University  
[conte@eos.ncsu.edu](mailto:conte@eos.ncsu.edu)*

## Abstract

*We advocate using performance bounds to guide code optimizations. Accurate performance bounds establish an efficient way to evaluate benefits as well as overheads of code transformations without actually performing instruction scheduling. In this paper, we introduce a novel bound-guided approach to systematically regulate code size related instruction level parallelism (ILP) optimizations including tail duplication, loop unrolling and if-conversion. Our approach is based on the notion of code size efficiency, which is defined as the ratio of ILP improvement over code size increase. With such a notion, we can 1) develop a general approach to selectively perform optimizations to maximize the ILP improvement while minimizing the cost in code size; and 2) define the optimal tradeoff between ILP improvement and code size overhead and develop a simple heuristic to achieve this optimal tradeoff. Experimental results using SPEC CINT 2000 benchmarks show that the performance improves significantly with very little code size increase using our systematic way to regulate code transformations and the simple heuristic is effective in achieving the optimal tradeoff.*

## 1. Introduction

Compile time code optimizations have been proven to be very successful in boosting the performance of microprocessors. However, various optimizations or the same type of optimizations performed at different locations of a program may have different performance impact. In order to use code optimizations efficiently and judiciously, two major issues need to be addressed. First, an

effective cost-benefit model is needed to analyze/predict performance gains without actually performing the time-consuming instruction scheduling or even the optimization itself. Secondly, a systematic way is desirable to selectively apply various types optimizations based on the cost model. We advocate using performance bounds as an indication of an optimization's effectiveness. Using performance bounds enables us to measure the performance limit of an optimization and also helps us to understand the bottlenecks when the performance potential is not fully achieved.

In this paper, we focus on code size related instruction level parallelism (ILP) optimizations including tail duplication, loop unrolling, and if-conversion. These optimizations are commonly used in forming and enlarging a scheduling region and have been shown to achieve big performance improvement [1, 3, 10, 29]. We introduce a novel approach to systematically regulate these optimizations. Our approach is based on the notion of code size efficiency, which is defined as the ratio of ILP improvement over code size increase. With this quantitative measure, we propose an algorithm to regulate code size related optimizations. The algorithm examines code size reducing transformations first. Then, it sorts instances of code enlarging optimizations, e.g., loop unrolling and tail replication at different parts of a program, based on their efficiencies. Next, an iterative approach is used to selectively perform these optimizations following their efficiency order until the overall program size or the size of hot portions of a program reaches a predetermined limit (e.g., I-cache size).

The code size efficiency also enables us to define the optimal tradeoff between the ILP improvement and the code size increase. A simple heuristic is then developed to achieve this optimal tradeoff. Experiments using SPEC CINT 2000 benchmarks [21] show that the performance improves significantly with very little code size increase using our systematic way to regulate code transformations and the simple heuristic is effective and robust in achieving the optimal tradeoff.

The remainder of the paper is organized as follows. Section 2 discusses background work. The code size efficiency is defined in Section 3. The algorithm to regulate code size related optimizations is described in Section 4. Section 5 defines the optimal tradeoff between performance improvement and code size and develops a simple heuristic to achieve this optimum. Section 6 contains the experimental methodology and results. Finally, Section 7 concludes the paper.

## 2. Background

Performance bounds have been proposed for many different purposes. In [9], a set of performance bounds are applied to scientific workloads to evaluate a range of computer architectures. These techniques are proven to be effective to gain insight of performance inhibitors when achieved performance fails to reach the performance bounds. In [22, 23], tight lower bounds of basic block scheduling are computed to guide hardware synthesis. Lower bounds for superblock scheduling is introduced in [13] and used as a heuristic of superblock scheduling. Our performance bound calculation in Section 3, is similar to these previously introduced bounds essentially since all these bounds are trying to capture the data dependence and resource constraints impact. Our bound is defined for a single-entry multi-exit region and the purpose is to estimate the performance potential of a code transformation.

A great number of code transformation techniques have been proposed in the literature. As our target in this paper is code size related optimizations in integer workloads, we focus on three most commonly used ILP optimizations: tail duplication, loop unrolling and if-conversion.

Tail duplication (or code replication) replicates a subgraph of control flow to remove side entries of a trace [1, 8]. Another benefit is to avoid the conditional / unconditional branches [5]. Many instruction-scheduling approaches [1, 3, 6] involve tail duplications in forming a scheduling region. Due to its evident impact on static code size increase, different heuristics have been proposed to decide

whether a particular tail-duplication should be performed. One simple example is a threshold on the profiled execution frequency [1]. However, there lacks a systematic way to analyze the tradeoff between the cost in code size and the performance gain.

Loop unrolling is another technique to enlarge a scheduling region. Modulo scheduling [11] may also benefit from it to reach a non-integer MII [18]. However, it has been recognized that loop unrolling can degrade performance if it is not used judiciously due to the increased code size and the increased resource requirements. In [14], Sarkar proposed a mechanism to automatically select unroll vector for nested loops. His approach associates a cost model for each feasible unroll vector and the one with best objective function is selected. The cost model evaluates the unroll vector without performing the unrolling. It uses a similar approach to minimum initiation intervals (RecMII and ResMII) computation as used in modulo scheduling to estimate the ILP for a candidate unroll vector. In [20], an iterative compilation approach is proposed to search the best unroll factor and tile size. Instead of a cost model, the actual execution time on the target machine is used. While these approaches are mainly targeted at scientific codes, our focus is integer workload.

If-conversion [24, 25] replaces conditional branches with appropriate predicate computations and the instructions that are control dependent on the branch are guarded with predicates. The removal of frequently mispredicted branches can yield large performance gain [26]. Also, if-conversion increases the spatial locality of instructions and may reduce code size if the targeted ISA requires the predicate computation for a conditional branch, such as IA64 [10, 27] or HPL-PD [2]. As pointed out in [19], full if-conversion generally works for compiling numerical application. For integer applications, selective if-conversion is essential to achieve performance gains due to the potential hazards of if-conversion. Hyperblock formation involves a complex heuristic to choose which paths to be included and then perform the if-conversion to the selected basic blocks [3]. The profile based selective if-

conversion proposed in [15] uses profile to compute the weighted schedule estimates before and after predicating a hammock. The schedule estimates are based on the local schedule results. Our performance bound calculation is more accurate as it takes consideration of potential control speculation in the hammock.

Note that all these optimizations have been proven to be very effective. The purpose of this paper is not to reiterate the importance of these optimizations. Instead, our objective is to advocate using performance bounds to evaluate the potential performance improvement and to introduce a systematic way of regulating these optimizations so that performance gains are maximized and the cost in code size is minimized.

As a final point, we proposed code size efficiency in [12] and showed its effectiveness in selectively performing tail duplication. The resulting performance benefits not only from the improved ILP but also from the increased spatial locality (better I-cache performance). This paper extends the idea as a general approach to regulate code size related transformations and highlights the importance of using performance bounds to evaluate the potential of an optimization.

### **3. Performance Bound Driven Code Size Efficiency**

In this section, we first define the notion of code size efficiency (CSEF). Then, we use tail duplication, loop unrolling, and if-conversion to explain how to use performance bounds to calculate the efficiency.

#### **3.1. Code size efficiency**

As the major objective of code size related optimizations is to improve instruction level parallelism (ILP), one direct measure of the effectiveness of such a transformation is to use the ratio of ILP improvement over the code size increase. Since code optimizations are performed at compile time, we

use static instruction-per-cycle (IPC) to measure ILP improvement. The static IPC is computed as the ratio of the number of retired instructions (IC) over estimated execution time (CT). Both IC and CT are derived from profile information and the speculated instructions resulting from instruction scheduling are not included in IC. Using the ratio of ILP improvement over code size increase as a quantitative measure, we can have two formal definitions of code size efficiency for code transformations.

First, we define the efficiency for an instance of a code transformation and we call it instance code size efficiency, as shown in Equation 1:

$$Efficiency_{instance} = \frac{IPC_{after\_individual\_application} - IPC_{before\_individual\_application}}{code\_size_{after\_individual\_application} - code\_size_{before\_individual\_application}} \quad (1)$$

In Equation 1, the term in nominator represents the ILP improvement of a particular instance of a code optimization and the term in denominator represents the cost of such an optimization in terms of static code size. Using the loop unrolling as an example, if we unroll a particular loop once, the instance efficiency of such an unrolling is the performance gain divided by the size of the loop body. Since there could be many loops in a program, there is an instance efficiency associated with each of them.

The definition in Equation 1 measures the performance impact at the cost of unit code size increase for a single instance of a code transformation. It is also useful to have a quantitative measure when more than one optimization instance have been performed. For example, assume a program has three loops. One unroll heuristic picks all three of them to be unrolled once and another heuristic may unroll just one loop many times. A quantitative measure would be able to tell which heuristic performs better in balancing the performance and code size. Such a measure is what we define as average code size efficiency, shown as Equation 2.

$$Efficiency_{average} = \frac{IPC_{candidate} - IPC_{original}}{code\_size_{candidate} - code\_size_{original}} \quad (2)$$

Similar to Equation 1, average efficiency measures performance gains in terms of ILP improvement at the cost of code size increase. The difference is that Equation 1 is used to evaluate an individual optimization instance and Equation 2 is used for the combined impact of many instances, of the same or different optimizations. In fact, average efficiency can be viewed as averaging the instance efficiencies of each individual code optimization that has been performed.

Note that the IPC improvement in Equations 1 and 2 closely correlates to the execution time reduction. In fact, we may use the ratio of execution time reduction over code size change to approximate code size efficiency (the difference between this ratio and the formal efficiency definition is a near constant factor for a given program). This ratio is easy to understand and intuitively appealing as it basically tells how many cycles can be saved with the cost of one additional instruction.

### 3.2. Using performance bounds to calculate code size efficiency

As shown in Equations 1 and 2, the ILP improvement of code optimizations is measured using static IPC, which involves two terms, IC and CT. IC is computed using block and edge profile information and remains constant as further increase/decrease of instructions due to code transformations and instruction scheduling are not counted. CT, however, varies (hopefully decreases) as a result of code transformations. To calculate the actual CT reduction, the scheduled code is needed, which implies we need to perform instruction scheduling to evaluate the actual impact of a transformation. As instruction scheduling is time consuming, such an approach is not practical. As discussed in Section 2, in practice various heuristics are used to estimate benefits instead of performing instruction scheduling. Our approach is to use performance bounds to evaluate the effectiveness of an optimization by how much the bounds are reduced.

As many compiler frameworks use a multi-path region as a scheduling unit, we establish a lower bound for a single-entry multiple-exit region:

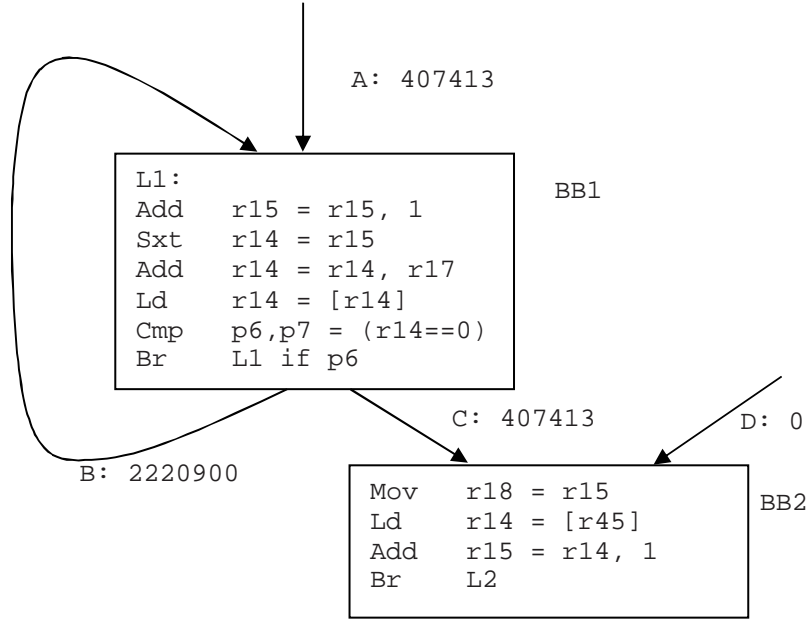
$$LBET = \sum_{path\_i} Max(data\_dependence\_bound_{path\_i}, resource\_bound_{path\_i}) * freq_{path\_i} \quad (3)$$

In Equation 3, the *lower bound of execution time* (LBET) of a region is computed as a weighted sum of the lower bound execution time of each path, which is in turn computed as the maximum of the *data dependence bound* and the *resource bound* of the path. True data dependence height of Data Dependence Graph (DDG) is used as the dependence bound assuming software renaming is available at schedule time. By calculating the data dependence bound along each path, the potential control speculation effect is considered implicitly as control dependence is not enforced. Resource bound in Equation 3 is calculated using a technique quite similar to ResMII calculation in iterative modulo scheduling [11]. The execution frequency for each path,  $Freq_{path\_i}$ , is obtained from edge profiling.

### 3.3. Examples of code size efficiency computation

First we focus on code transformations resulting ILP improvement as well as code size increase. Both tail duplication and loop unrolling are such optimizations. Using a code segment in the benchmark *twolf* as an example, shown in Figure 1, we explain how to compute the code size efficiency.

The code segment shown in Figure 1 has two basic blocks (BB1 and BB2) and there exist a loop back edge (edge B) and a merge point (edges C and D), exhibiting the possibility of applying both loop unrolling and tail duplication.

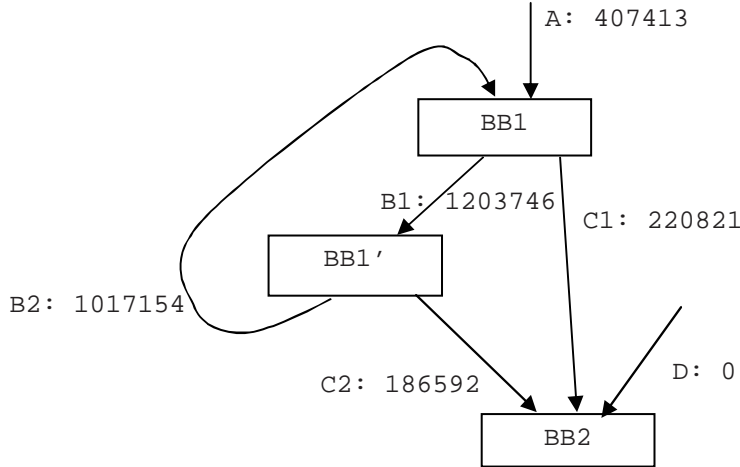


**Figure 1. Code segment of twolf (in function new\_dbox\_a). Numbers along control edge labels are edge profiles.**

Assuming load instructions have 2-cycle latency and all other instructions in BB1 and BB2 have 1-cycle latency (except CMP instructions which can be scheduled at the same cycle as the consuming branch), the lower bound execution time (LBET) before any transformation is the sum of LBET of BB1 and LBET of BB2. Assuming a 6-wide issue (2 ALU, 2 ALU/LD/ST, 2 ALU/BR) machine (which causes no resource constraints in this example), LBET can be computed using Equation 3: LBET of BB1 is  $6 \cdot 2628313 = 15769878$  cycles, LBET of BB2 is  $3 \cdot 407413 = 1222239$  cycles, and the sum is 16992117 cycles. After duplicating of BB2, the instructions in BB2 can be scheduled in BB1 using control speculation, which results in  $\text{LBET} = 15769878$  cycles as the inclusion of BB2 instructions does not increase the true data dependence height (i.e., LBET reduction of 1222239 cycles due to complete hiding of BB2 execution time). Therefore, the instance code size efficiency of tail duplication happening at merge point of edge C and D is  $1222239 / 4 = 305560$  cycle/instruction, i.e., one additional instruction leads to 305560 cycle execution time reduction.

Similarly, we can compute the efficiency of unrolling the loop body in Figure 1, i.e., BB1. As the loop-carried dependence height in this example is 1 cycle, original loop body can overlap much of the

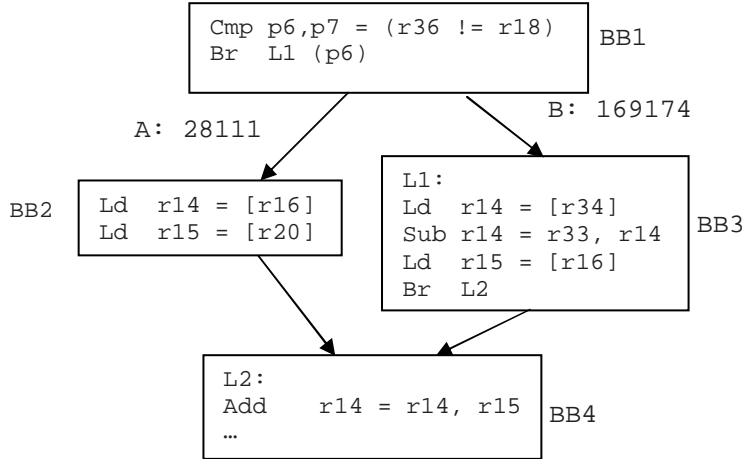
computation with the unrolled copy. Here, we need to be careful in distributing profile data after loop unrolling. The probability propagation approach proposed in [17] is used in this paper and the results of unroll BB1 once is shown in Figure 2.



**Figure 2. Loop unrolling of loop body shown in Figure 1 with unroll factor as 1. (Numbers along control edge labels are edge profiles computed using probability propagation.)**

As shown in Figure 2, the probability propagation maintains the taken/not taken probability of the conditional branches at the end of BB1 and BB1' (the unrolled copy of BB1). After the profile is redistributed, the LBET of loop body in Figure 2 (containing BB1 and BB1') can be computed using Equation 3 (9751148 cycles). Compared to LBET of loop body with no unrolling, LBET reduction is  $15769878 - 9751148 = 6018730$  cycles. Therefore, the instance code size efficiency of loop unrolling (with factor 1) at back edge B is:  $6018730 / 6 = 1003121$  cycles/instruction.

If-conversion can *reduce* code size by removing branch instructions. Also, it may result in speedups by removing branch misprediction penalties. Therefore, the code size efficiency can be a negative number (i.e., positive speedup and negative code size increase), which represents one highly desired extreme of code size efficiency. (The other extreme of negative speedup and positive code size increase is what we always want to avoid.) Using another simple code segment in the benchmark *twolf*, we show how we compute the efficiency of if-conversion by integrating branch misprediction penalties. The code segment is shown in Figure 3.



**Figure 3. Code segment of twolf (function add\_penal) to show efficiency of if-conversion. Numbers along control edge labels are edge profiles.**

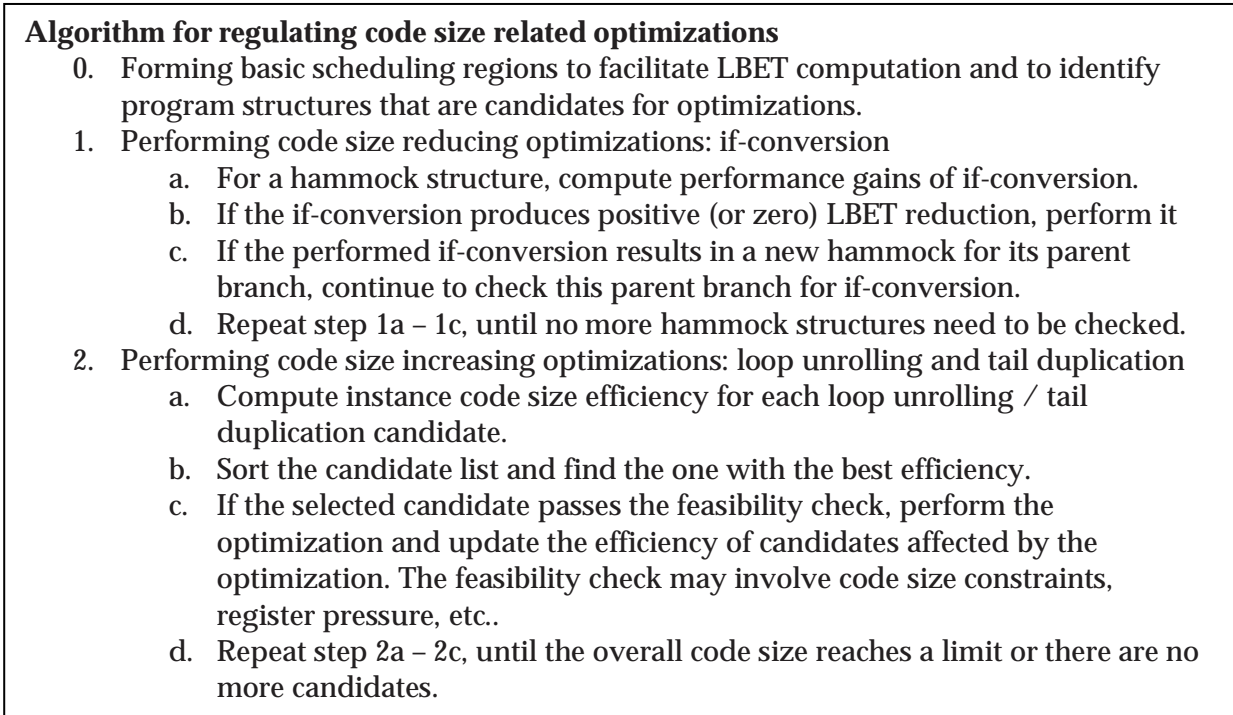
Using Equation 3, LBET of the region containing BB1, BB2 and BB3 is computed as  $28111*2+169174*3 = 563744$  cycles. Then, we consider potential branch misprediction penalties. Assuming static branch prediction and a 10-cycle misprediction penalty for each misprediction, the overall misprediction penalty of this conditional branch in BB1 is  $28111*10 = 281110$  cycles. If the profile of dynamic branch prediction is available, more accurate penalty computation can be used.

After if-conversion, the branches in BB1 and BB3 are removed (i.e., 2 instruction reduction) and the resulting LBET is  $3*(28111+169174) = 591855$  cycles, which means a reduction of  $(563744+281110-591855) = 252999$  cycles. Note that this computation involves only the control dependent blocks of the conditional branch (BB1, BB2 and BB3). It does not depend on the merge basic block (BB4) and the same result holds when BB4 has more than 2 entry edges.

As pointed out previously, optimizations with positive speedup and negative code size increase are always performed. So, we do not need to calculate the actual efficiency for such cases. For if-conversions that have both negative speedup and negative code size increase, positive code size efficiency is resulting. Such efficiency has an implication that we may want to perform if-conversion with low positive efficiency to reduce code size (although hurting performance slightly) and use the saved code size for optimizations with higher efficiency.

#### 4. Regulating Code Size Related Optimizations

Based on the quantitative measure of code size efficiency defined in Section 3, we can develop a general algorithm to regulate code size related optimizations, as shown in Figure 4



**Figure 4. Algorithm for regulating code size related optimizations**

The algorithm in Figure 4 has three steps in regulating different kinds of code size related code transformations. As a preparation step, we construct basic scheduling regions without performing any region-enlarging optimizations. The examples are treeregion formation without tail duplication (i.e., the natural treeregion [7]) or superblock formation without tail duplication. Such regions are single-entry multiple-exit regions for which LBET can be computed using Equation 3.

Optimizations are treated differently based on their code size efficiency characteristics. Optimizations with positive speedup and negative code size increase are examined first in Step 1 in the algorithm. Then, an iterative approach is used to selectively performing code-expanding optimizations, as shown in Step 2 of the algorithm. First, step 2a computes the efficiency of all potential optimization instances. Then, these instances are sorted based on their efficiency in step 2b. Next, if the one with the

best efficiency passes the feasibility check, it will be performed in step 2c. The feasibility check basically makes sure a particular optimization will not result in excessive resource utilization, e.g., the size of a loop body is less than level one I-cache size. As one particular optimization may change the efficiency of another optimization or enable another optimization (e.g., a tail duplication may enable a hammock to be constructed for if-conversion), local efficiency update is performed in Step 2c if one optimization instance is performed. Note that this iterative approach can automatically choose a good unroll factor for a loop by unrolling the original loop body one iteration a time.

### 5. Optimal Tradeoff between ILP Improvement and Code Size Increase

For code size increasing optimizations, the algorithm in Section 4 iteratively selects and performs those with best code size efficiency. If we use a curve to represent the resulting ILP improvement and relative code size increase, which we call the *ILP vs. code size curve*, a very interesting phenomenon reveals: optimizations among initial picks exhibit large performance improvement with small code size increase (i.e., high efficiency) and those selected later on show quickly decreasing performance improvement with relatively larger code size increase (i.e., low efficiency). Such a phenomenon is the ‘*diminishing returns*’, as we can see from Figure 5.

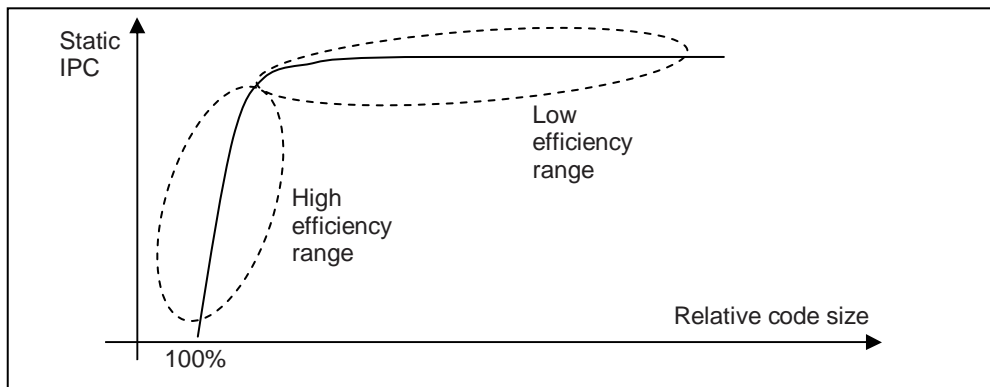


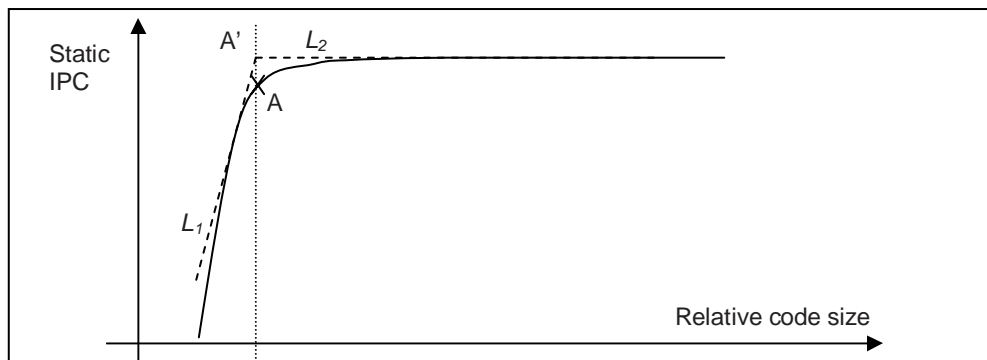
Figure 5. An example curve showing the relationship of ILP improvement and code size increase

Figure 5 shows an example ILP vs. code size curve, which exhibits common characteristics of individual benchmarks we studied (see Section 6.3). The diminishing returns are due to the rapidly

decreasing code size efficiencies, which in turn is due to the following two reasons. First, based on the definition of code size efficiency, the instance optimization with high efficiency should have high execution frequency. The well-known 90/10 rule points out that a small part of the static code (hot portion) consumes over most of the execution time (we verified this in our experiments as well). After performing optimizations in this hot portion of code, the remaining optimizations should have much lower efficiencies due to the much lower execution frequency. Secondly, high efficiency also requires that the resulting code must have better performance bound, i.e., the instance optimization must reduce the DDG height without causing any resource conflict problem. This requirement filters the optimizations happening in hot portions of a program.

The diminishing returns phenomenon shown in Figure 5 enables us to define the optimum tradeoff between ILP improvement and code size increase. The natural choice is the ‘knee’ of the curve in Figure 5, assuming that the corresponding code size still satisfies the feasibility check.

To automatically find this knee of curve, a simple heuristic is developed by taking advantage of the steep slope of the high efficiency part of the ILP vs. code size curve, as shown in Figure 6. Figure 6 replicates the ILP vs. code size curve in Figure 5 and the knee of the curve is marked as point A.



**Figure 6. Achieving the optimum tradeoff between ILP improvement and code size increase**

To locate A, we can first use two straight lines to approximate the curve (as the dashed lines  $L_1$  and  $L_2$  shown in Figure 6). Then, the knee of the curve becomes point A'. A simple threshold scheme can be used to find A': the point along the curve whose slope is between slope of  $L_1$  and slope of  $L_2$ . The

slope of the ILP vs. code size curve represents the ratio of static IPC changes over relative code size changes, which is the definition of the instance code size efficiency in Equation 1. So, the approach to achieve the optimum tradeoff is simply as follows: *performing the optimizations whose instance code size efficiency is higher than the threshold efficiency  $K$* . This threshold efficiency can be any value between the slope of  $L_1$  and slope of  $L_2$ . In other words, the range between slope between  $L_1$  and slope of  $L_2$  determines the *robustness* of this threshold scheme. In our experiments (see section 6.4), we vary  $K$  from  $\tan(\pi/12)$  (corresponding to a line with an angle of 15 degrees) to  $\tan(\pi/6)$  (corresponding to a line with an angle of 30 degrees) to show the robustness of this threshold scheme.

As we keep using the ratio of LBET change over absolute code size increase (measured in number of instructions) to compute code size efficiency, we can further derive the threshold scheme as in Equation 4. The derivation details can be found in an earlier technical report [28].

$$\frac{d(-LBET)}{dSize_{absolute}} \geq \frac{K * LBET}{IPC_{static} * IC_{static}} \quad (4)$$

In Equation 4,  $IC_{static}$  represents the static operation count of the program (i.e., the static program size; whereas the term  $IC_{dynamic}$  is the number of retired instructions during execution and is used for IPC calculation),  $K$  is the threshold on instance code size efficiency,  $LBET$  is the lower bound of execution time of the whole program,  $d(-LBET)$  is the reduction in the lower bound (both are computed using Equation 3), and  $IPC_{static}$  ( $= LBET / IC_{dynamic}$ ) represents the ILP feature of the original program.

## 6. Experimental Results

In this section, we first describe our experimental methodology and present results using the algorithm in Section 4. Then, we show the effectiveness and robustness of the threshold approach in Section 5.

## 6.1. Methodology

In our experiments, we use SPEC CINT 2000 benchmarks to evaluate the proposed algorithms. The benchmarks are first compiled into IA64 assembly using *gcc* compiler (version 3.1). As our purpose is to regulate ILP optimizations, we use level one optimization of *gcc* to perform the classical optimization (as discussed in Section 6.2, a by-product of level one optimization is that *gcc* also produces predicated code). The resulting IA 64 assembly codes are then parsed into our LEGO compiler framework [16], which we use to implement the algorithms in this paper. The IA 64 assembly is instrumented and executed to gather the profile information. In our experiments, we use reference input data set and skip first 500 million instructions and profile next 500 million instructions.

Treeregion-based instruction scheduling [6, 7] is used in LEGO compiler. In treeregion scheduling, a treeregion, which is a single-entry multiple-exit sub-graph of control flow graph (CFG) of a program, is the basic scheduling region. We first use natural treeregions (treeregions formed without any tail duplication) to get the baseline execution time and static IPC. Performance bounds calculated using Equation 3 are used as baseline execution time, which represents best schedule achievable without any further optimization. The baseline results are show in Table 1, which include static code size, number of dynamic retired instructions (around 500M as we profiled 500M instructions) and lower bound of execution time. Static IPC indicates the inherent ILP present in the current code and the results show that many benchmarks have moderate ILP (IPC around 2) while the benchmark *gap* has very limited ILP (IPC around 1). An examination of the benchmark *gap* finds that the function *ProdInt* is heavily executed in our profile phase. The complex computations (long dependence chain) in this function result in low ILP.

In Table 1, we also include the ratio of estimated execution time of treeregion-scheduled code over the lower bound. The execution time of treeregion-scheduled code is computed using a scoreboard

dependency-enforcing approach (i.e., it is the execution time assuming ideal cache and ideal branch prediction). From those results, it can be seen that the treegion scheduler produces a quite good schedule, exceeding 1% to 13% of the lower bound. The mismatch is because the performance bound is calculated assuming that all false register dependencies can be removed by software renaming, and that control dependencies can be minimized by multiway branch transformation. Such assumptions are too optimistic as liveness beyond basic block scope may require a copy instruction to be inserted. Resource confliction due to speculation from multiple paths in a treegion is another reason.

**Table 1. Baseline results including static code size, execution time, and static IPC**

Baseline	bzip	crafty	Gap	gzip	mcf	parser	twolf	vortex	vpr
Static size (num of insn)	7543	51085	131447	13316	2548	25545	65786	120735	35416
Number of dynamic insn retired	498M	490M	500M	495M	491M	49M	496M	499M	497M
Lower bound of exe. time (cycles)	257M	217M	495M	275M	276M	263M	325M	219M	318M
Static IPC	1.93	2.26	1.01	1.80	1.78	1.87	1.53	2.27	1.56
Ratio of natural tree schedule results over the lower bound	104%	108%	104%	112%	106%	113%	107%	107%	101%

## 6.2. Regulating code size decreasing optimizations – if-conversions

Step 1 of the algorithm shown in Figure 4 regulates how code size decreasing optimizations, if-conversion in this paper, are performed. Due to its code size reduction effect, any if-conversion, which produces positive (or zero) LBET reduction (i.e., positive speedups), will always be performed. As described in Section 3.3, we use static branch prediction to estimate branch misprediction penalties assuming that each misprediction associates a 10-cycle penalty.

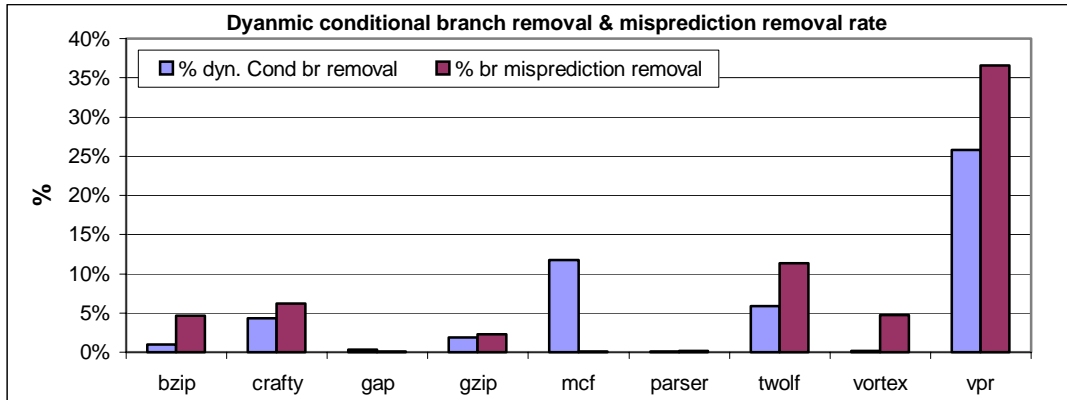
Table 2 shows the if-conversion results using our algorithm. As stated previously, the input IA64 assembly code is generated using GNU *gcc* compiler with level one optimization, which performs not only classical optimizations but also if-conversion. By applying our algorithm to this already if-converted code, we basically show that our algorithm can improve *gcc*'s if-conversion algorithm.

**Table 2. If-conversion results**

	bzip	crafty	gap	gzip	mcf	parser	twolf	vortex	vpr
If-conversions (by gcc)	113	780	2852	139	61	502	1042	1692	325
Number of conditional br.	487	2712	9747	819	167	2068	3625	7469	1805
If-conversion with pos. gain	2	40	1	6	5	2	11	4	10
If-conversion with zero gain	19	163	324	26	7	80	445	191	74
If-conversion with neg. gain	4	58	2	3	4	1	24	37	8
No if-conversion: complex CFG	358	1608	5752	546	133	1483	2787	2161	1263
No if-conversion: ret_call	104	843	3668	238	18	502	358	5076	450
Number of dynamic cond. br.	36.4M	23.8M	23.6M	37.4M	71.0M	46.0M	33.8M	32.8M	30.5M
Reduction in Execution time (cycles)	91016	1057363	9700	799877	90688	80125	506372	122592	13148695
static br. misprediction rate	7.34%	12.78%	11.61%	10.42%	15.44%	12.03%	13.40%	0.92%	14.35%

Interesting observations can be made from Table 2. The first row in Table 2 reveals that *gcc* has removed a significant amount of conditional branches through predication, although the second row, which shows the number of existing conditional branches in each benchmark after *gcc*'s if-conversion, suggests that there still exist potential if-conversion candidates. Our algorithm examines those conditional branches and confirms that the majority of these conditional branches are hard to if-convert. We report those hard-to-convert conditional branches in two categories: row 6 shows the number of conditional branches followed with complex CFG (e.g., merging points at both if path and else path of a hammock) inhibiting hammock detection, and row 7 presents the number of the detected hammocks containing either function call, return, or indirect branch instructions (we excluded the case that both paths contain the same function call or return instruction). For those if-convertible branches, our algorithm computes the performance gain. Using the benchmark *gzip* as an example, *gcc* converts 139 conditional branches and there remain 819 conditional branches in the program. Our algorithm finds that 546 of them cannot form a hammock structure. For those that form a hammock, 238 of them have either a function call or a return along the paths. For the remaining ones, 6, 26, and 3 of them produce positive, zero, and negative speedups, respectively.

Next, we analyze the performance impact of if-conversions. In this experiment, we perform only the if-conversions that produce positive gains. Although the number of these if-conversions seems limited (1 to 40), significant performance gains can be achieved, as shown in Figure 7.



**Figure 7. The removal rate of dynamic conditional branches and mispredictions by if-conversion**

Figure 7 shows the percentage of dynamic conditional branches and associated mispredictions removed by if-conversions. It can be seen that 0.1% (*parser*) to 25.8% (*vpr*) of dynamic conditional branches can be eliminated and 0.1% to 36.6% branch mispredictions associated with these conditional branches can be removed, assuming static branch prediction. Note that higher rate of dynamic branch removal does not necessarily mean higher reduction in mispredictions. For example, if-converting 5 conditional branches in the benchmark *mcf* reduces the number of dynamic conditional branches by 12%, which only results in 0.1% reduction in branch mispredictions. The reason is that the removed conditional branches are highly biased, which in turn produces small reduction in LBET as shown in row 9 of Table 2. For completeness, the number of dynamic conditional branches is shown in row 8 and static branch misprediction rates for conditional branches are included in the last row of Table 2.

At last, we analyze the code size reduction impact of if-conversions. We choose to perform if-conversion with positive or zero gains in this experiment. Assuming each conversion saves two instructions in IA64, the overall code size reduction is computed and is shown in Figure 8. Remember that this reduction is achieved on the IA64 codes that have already been predicated by *gcc*. So, it

basically shows that our algorithm performs if-conversion more aggressively in reducing code size. From Figure 8, it can be seen that if-conversion reduces up to 1.4% of static code and 0.68% in average. Although these numbers seem to be trivial, in next subsection, we will show that utilizing such a small amount of code size can produce very large ILP improvements.

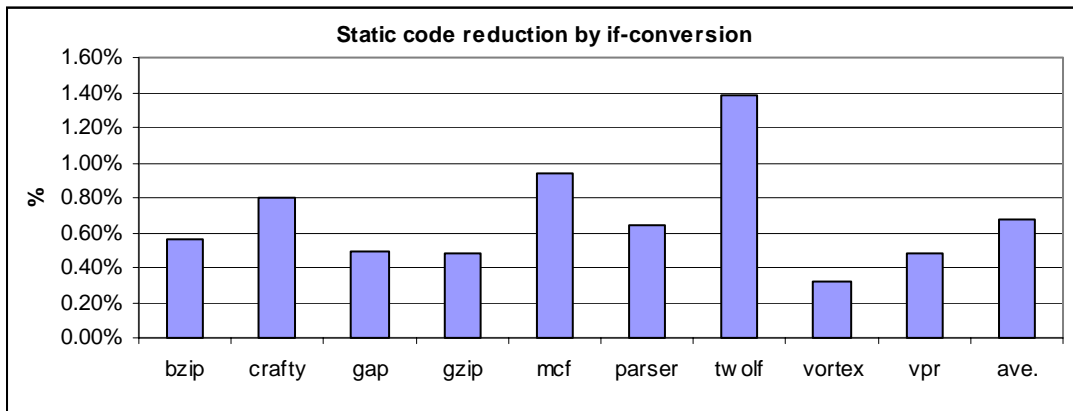


Figure 8. The static code size reduction by if-conversion

### 6.3. Regulating code size increasing optimizations – tail duplication and loop unrolling

Step 2 of the algorithm shown in Figure 4 regulates code size increasing optimizations, tail duplications and loop unrolling in this paper. It iteratively selects and performs one instance of tail duplication or loop unrolling based on its instance efficiency. In this experiment, we examine the effectiveness of such an iterative approach. For each benchmark, we set the limit of overall code size increase as 1%, 2%, and 5% of its original size (i.e., the optimization stops when the overall code size reach the limit). The corresponding ILP improvements are shown in Figure 9.

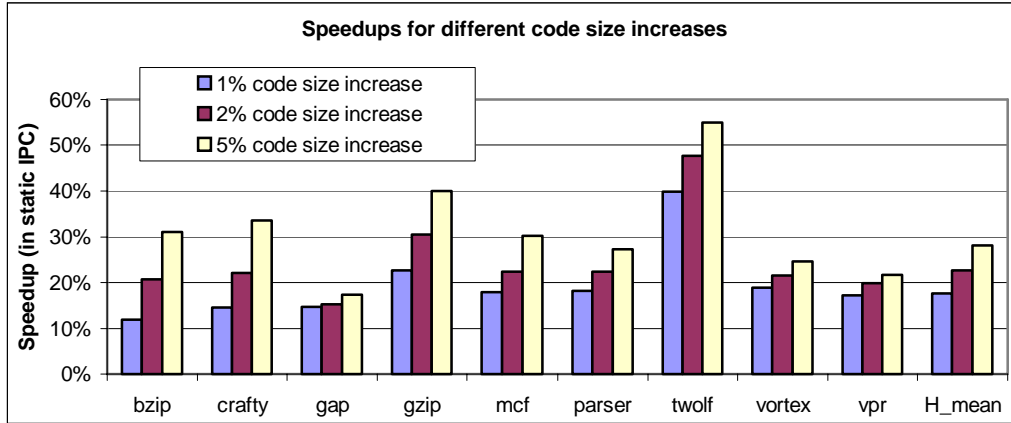
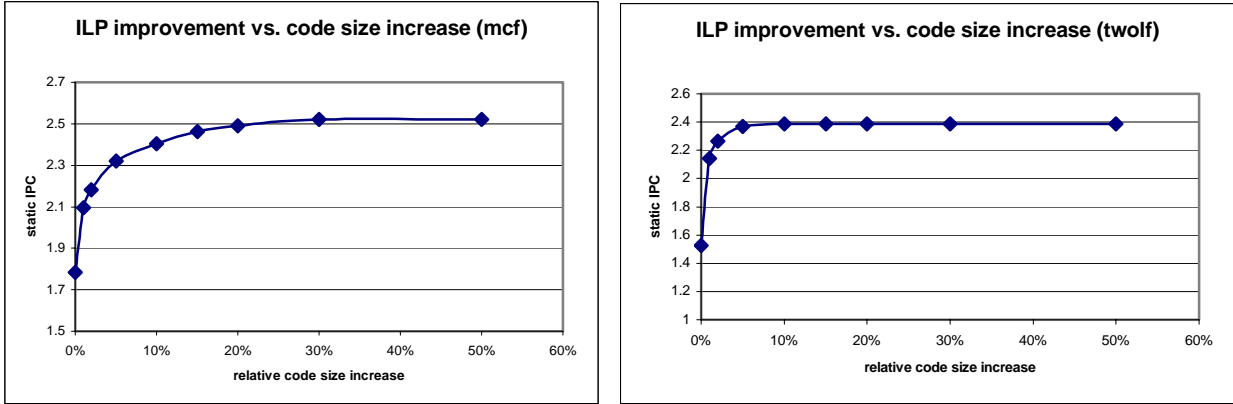


Figure 9. The speedups for different code size increases

Two major observations can be made from Figure 9. First, it is evident that a very small amount of code size increase can lead to significant improvement in ILP if this code size is used judiciously. Our algorithm achieves up to a 40% increase and an average of 18% increase in static IPC when the code size is expanded by just 1%. This, on the other hand, emphasizes the importance of code size reducing optimizations. The code size saved by aggressively performing code-reducing transformations can then be used for code expanding optimizations with high efficiency. This is the reason that the algorithm in Figure 4 performs code-reducing transformations before code enlarging ones. Secondly, it can be seen from Figure 9 that further code size increase has less impact on ILP improvement. As shown in the figure, an additional speedup of 5% in average is observed as the code size increases from 1% to 2% of its original size, still significant but less impressive as 18% for the first 1% code size increase. The reason is that during the iterative selection process, the efficiency of the selected optimization decreases rapidly. Using the benchmark *vortex* as an example, the first selected optimization is one tail-duplication in procedure `Chunk_ChkGetChunk` with efficiency as high as 534,609 cycles/instruction. After following 7 optimizations were selected and performed (resulting replicating 66 instructions), the efficiency of the next chosen optimization decreases to 77,484 cycles/instruction. As discussed in Section 5, two main reasons account for such ‘diminishing returns’: the 90/10 rule and the reduction in performance bounds.

Next, two individual benchmarks, *mcf* and *twolf*, are chosen as representative benchmarks to examine the diminishing returns impact in detail. A curve of ILP vs. code size is shown for each benchmark in Figure 10.



**Figure 10. ILP improvement vs. code size increase for benchmarks mcf, twolf, and vortex**

In Figure 10, the code size increase of each benchmark is normalized to its original code size. The curve of ILP improvement vs. code size increase is obtained as follows. First, we set limit of relative code size increase as 1%, 2%, 5%, 10%, 15%, 20%, 30%, and 50% and use the iterative approach to selectively performing code size increasing optimizations. Then, we produce the curve by interpolating these results. From these two benchmarks, we can see that diminishing returns usually happen quickly with small code increase. For the benchmark *twolf*, it happens at around 5% code size increase while the benchmark *mcf* shows that the performance can still be improved significantly until the increase is around 20%.

#### 6.4. Achieving the optimum tradeoff between ILP improvement and code size increase

As discussed in Section 5, the diminishing returns that are observed in Figure 10 enable us to define the optimum tradeoff between ILP improvement and code size increase. Also, a simple threshold scheme is developed to achieve this optimum. In this experiment, we show the effectiveness as well as the robustness of this threshold scheme.

First, we examine the robustness of our scheme. We set  $K$  as  $\tan(\pi/6)$  and compute the threshold on instance code size efficiency for each benchmark using Equation 4, as shown in Table 3. Based on the threshold values, the optimizations with their efficiency exceeding the threshold are performed. The resulting code size and ILP improvement are also shown in Table 3. It can be seen that for many benchmarks, the resulting optimal tradeoff has a small code size increase (up to 18%) and a very large ILP improvement (up to 59%).

**Table 3. The resulting code size and ILP improvements when threshold  $K = 0.577$**

	bzip	crafty	gap	Gzip	mcf	parser	twolf	vortex	vpr
Efficiency threshold (cycles/instruction)	10211	1088	2153	6594	35022	3167	1867	461	3307
Resulting relative code size increase	18.2%	9.7%	1.5%	11.2%	17.1%	13.5%	4.8%	5.3%	5.5%
Resulting static IPC increase	59.4%	39.6%	15.0%	48.6%	38.0%	34.1%	55.9%	24.4%	33.1%

Then, we change  $K$  to  $\tan(\pi/12)$  and re-calculate the threshold values, as shown in Table 4. Compared to Table 3, it can be seen that the rather large change in threshold value (over 100%) results in very small variations in ILP improvement (up to 2.4%) and code size (up to 4.2%). This demonstrates the robustness of our threshold scheme.

**Table 4. The resulting code size and ILP improvements when threshold  $K = 0.268$**

	bzip	crafty	gap	Gzip	mcf	parser	twolf	vortex	vpr
Efficiency threshold (cycles/instruction)	4743	505	1000	3063	16267	1471	867	214	1536
Resulting relative code size increase	21.4%	13.8%	2.2%	13.4%	21.3%	18.6%	5.8%	7.5%	6.7%
Resulting static IPC increase	60.9%	40.9%	17.4%	49.7%	39.7%	35.7%	56.6%	24.9%	33.3%

Next, we use the representative benchmarks used in Section 6.3, i.e., the benchmarks *mcf* and *twolf* to show that our scheme does achieve the optimal tradeoff (i.e., the knee of the ILP vs. code size curve). Note that *mcf* shows the maximal variation in resulting code size for different thresholds, representing the worst case among all these benchmarks. The results are shown in Figure 11.

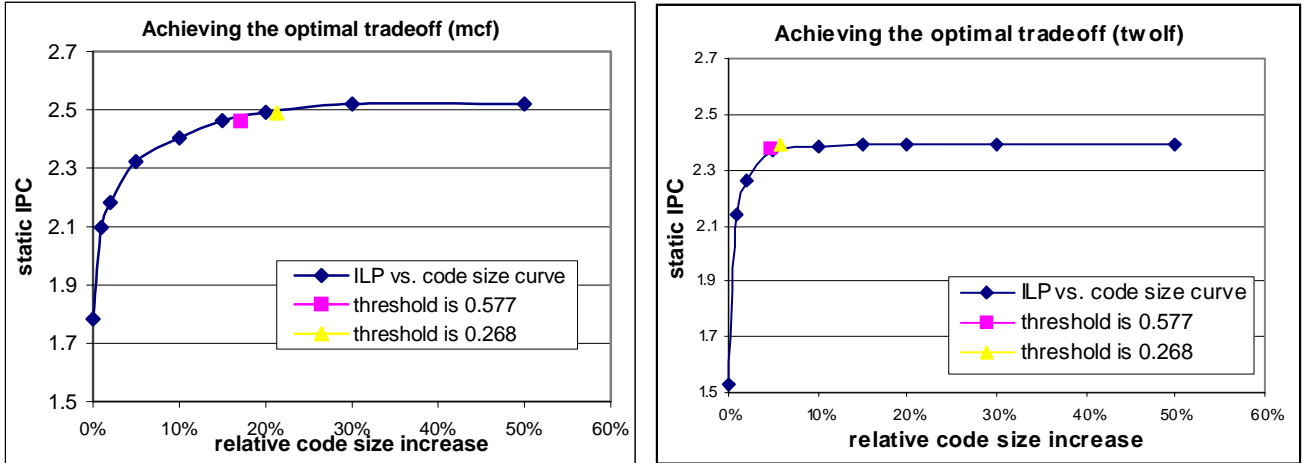


Figure 11. Achieving the optimal tradeoff between ILP improvement and code size increase (benchmarks *mcf* and *twolf*)

Figure 11 shows the tradeoff points obtained by our threshold scheme with different threshold values as well as the ILP vs. code size curve. The curve of the benchmark *twolf* shows a sharp turn around the knee and our algorithm finds the optimal trade off (or knee of the curve) precisely. For the benchmark *mcf*, the ILP vs. code size curve exhibits a less sharp turn around the knee. As a result, our algorithm generates two more distinct points along the curve. However, it can be seen that both points are still in the range of the ‘knee’. So, both of them are valid solutions.

## 7. Conclusion

This paper advocates using performance bounds to evaluate performance potentials of compile time optimizations. Based on a bound-driven notion of code size efficiency, a novel approach is developed to regulate code size related ILP optimizations in a systematic way. In this paper, three types of ILP transformations: if-conversion, loop unrolling, and tail duplication are considered. Our algorithm examines code size reducing optimizations first. Then, an iterative approach is used to selectively perform code-enlarging optimizations with the best efficiency. In such a way, maximal ILP improvement can be achieved with minimum code size increases. Experimental results using SPEC CINT 2000 benchmarks show that a very high ILP improvement (up to 40% and 18% in average) can

be achieved with a very small code size increase (1%). Considering the code size saved by if-conversion, the overall code size increase is further reduced (-0.4% to 0.7% overall increase).

In this paper, we show the interesting diminishing returns phenomenon in performing code-enlarging optimizations to improve ILP. The optimal tradeoff between the ILP improvement and code size increase can be defined as the knee of the ILP vs. code size curve. Then, a simple threshold scheme is developed to achieve this optimum. Experimental results demonstrate that our threshold scheme is effective and robust in achieving the optimal tradeoff.

## 8. Reference

- [1] 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.
- [2] 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.
- [3] 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. 25<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO25)*, December 1992.
- [4] 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.
- [5] F. Mueller and D. B. Whalley, "Avoiding Conditional Branches via Code Replication", ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1995
- [6] W.A. Havanki, S. Banerjia, and T. M. Conte. "Treeregion scheduling for wide-issue processors." *Proceedings of the 4<sup>th</sup> International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.
- [7] H. Zhou, M. Jennings, and T. M. Conte. "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors". *Proceedings of the 14<sup>th</sup> Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, LNCS, Springer Verlag, 2001.
- [8] D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling", *Proc. 24<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO24)*, 1991.
- [9] Bill Mangione-Smith, "Performance Bounds for Rapid Computer System Evaluation", in *Fast Simulation of Computer Architectures*, edited by Thomas M. Conte and Charles E. Givarc, Kluwer Academic Publishers, 1995.
- [10] Intel Corp, IA-64 Application Developer's Architecture Guide, 2000.
- [11] B. R. Rau, "Iterative Module Scheduling", Tech. Rep. HPL-94-115, Hewlett-Packard Laboratories, 1995.
- [12] H. Zhou and T. M. Conte, "Code size efficiency in global scheduling for ILP processors", 6<sup>th</sup> workshop on Interaction between Compilers and Computer Architecture, Feb. 2002.
- [13] A. E. Eichenberger and W. M. Meleis, "Balance Scheduling: Weighting Branch Tradeoffs in Superblocks", *Proc. 32<sup>nd</sup> Ann. Int'l Symp. Microarchitecture (MICRO32)*, 1999.
- [14] V. Sarkar, "Optimized Unrolling of Nested Loops", *Proceedings of International Conference on Supercomputing (ICS)*, 2000.
- [15] S. Mantripragada and A. Nicolau, "Using profiling to reduce branch misprediction costs on a dynamically scheduled processor", *Proceedings of International Conference on Supercomputing (ICS)*, 2000.
- [16] The LEGO Compiler. Available for download at <http://www.tinker.ncsu.edu/LEGO>.
- [17] Y. Wu and J. Larus, "Static branch frequency and program profile analysis", *Proc. 27<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO27)*, 1994.
- [18] D. M. Lavery and W W. Hwu, "Unrolling-based optimizations for modulo scheduling", *Proc. 28<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO28)*, 1995.

- [19] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication", *Proc. 30<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO30)*, 1997.
- [20] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle, "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation", *Proc. Of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT'00)*, October, 2000.
- [21] J. L. Henning, "SPEC CPU 2000: Measuring CPU performance in the new millennium", *IEEE Computer*, July 2000.
- [22] M. Langevin and E. Cerny, "A recursive technique for computing lower bound performance of schedules", *IEEE International Conference on Computer Design*, 1993.
- [23] M. Rim and R. Jain, "Lower-bound performance estimation for high-level synthesis scheduling problem", *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(4), 1994.
- [24] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence", *Proceedings of 10<sup>th</sup> ACM Symposium on Principles of Programming Languages*, 1983.
- [25] J. C. Park and M. S. Schlansker, "On predicated execution", Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, 1991.
- [26] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction", *Proc. 27<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO27)*, 1994
- [27] Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The impact of If-conversion and branch prediction on program execution on the Intel Itanium processor", *Proc. 34<sup>th</sup> Ann. Int'l Symp. Microarchitecture (MICRO34)*, 2001.
- [28] H. Zhou and T. M. Conte, "Code size efficiency in global scheduling for VLIW/EPIC style embedded processors", Technical Report, ECE Department, NC State University, 2002.
- [29] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: path based data representation and scheduling of subgraphs", *Proc. 32<sup>nd</sup> Ann. Int'l Symp. Microarchitecture (MICRO32)*, 1999.