Code Size Efficiency in Global Scheduling for ILP Processors

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

Abstract

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

1. Introduction

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

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

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

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

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

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

2. Treegion-based global scheduling and simulation methodology

2.1. Treegions and treegion-based global scheduling

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

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

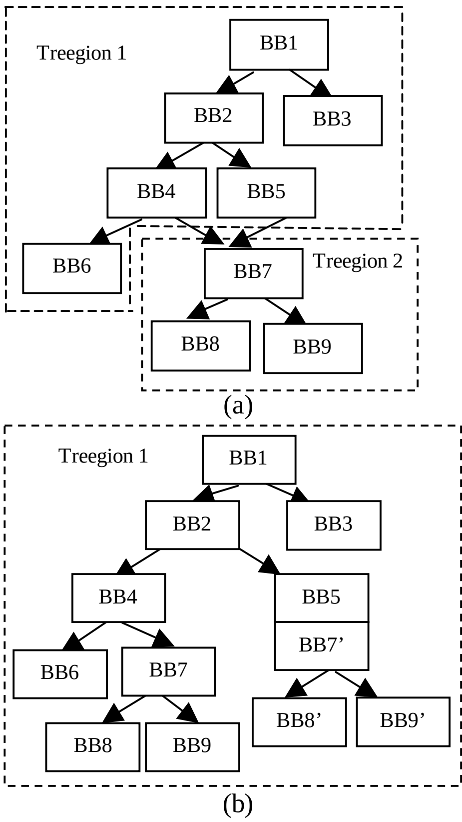

Figure 1. (a) The CFG and the treegions constructed; (b) The treegion constructed after the tail duplication

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

2.2. The code size increase in treegion scheduling

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

2.3. Simulation methodology

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

---

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

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

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

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

3. The quantitative measure of code size efficiency

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

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

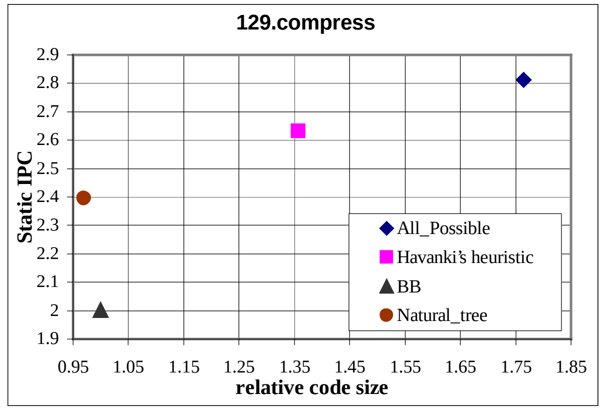

Figure 2. The relationship between performance and static code size for benchmark *compress*

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

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

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

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

In Equation (1), the term $(IPC_{candidate} - IPC_{natural\_treegion})$ represents the ILP improvement of the candidate optimizations and the term ($code\_size_{candidate} - code\_size_{natural\_treegion}$) represents the cost of such optimizations in terms of static code size. Graphically in Figure 2, the average code size efficiency represents the slope of a line connecting the natural treegion result and the one under consideration (i.e., 'candidate'). With this quantitative measure, the comparison can be made for different code size related optimizations and for the different applications of the same optimization. For example, based on tail duplication results in Figure 2, it can be seen that the Havanki's heuristic produces a slightly better code size efficiency than duplicating all the possible candidates. Note that if the efficiency of an optimization is calculated as negative, it represents one of two extreme cases: (a) the optimization increases the IPC and decreases the code size— this optimization should always be applied, or (b) the optimization decreases the IPC at the cost of more code size— this optimization usually needs to be avoided.

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

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

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

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

3.2. A heuristic to compute efficiency using expected execution time

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

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

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

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

3.3. The code size efficiency for tail duplication optimization

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

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

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

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

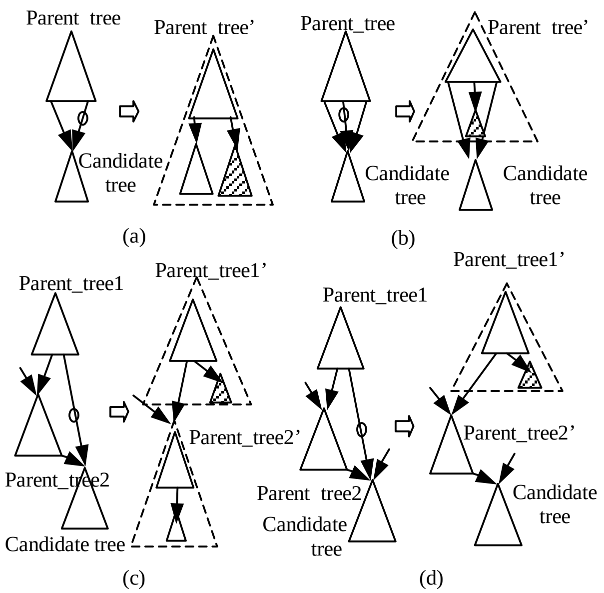

Figure 3. Four types of possible tail duplication in treegions (the edge marked with 'o' representing the edge to be removed by the candidate tail duplication, the shaded treegion represents the duplicated region): (a) Type-1: The parent tree dominates the candidate tree and there are 2 edges entering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

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

4. Optimal code size efficiency in global scheduling

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

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

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

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

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

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

Algorithm for optimal tail duplications under code size constraints

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

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

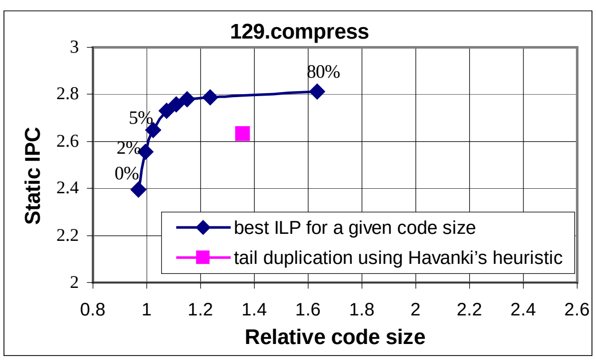

Figure 5. The relationship of ILP vs. code size of benchmark *compress*

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

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

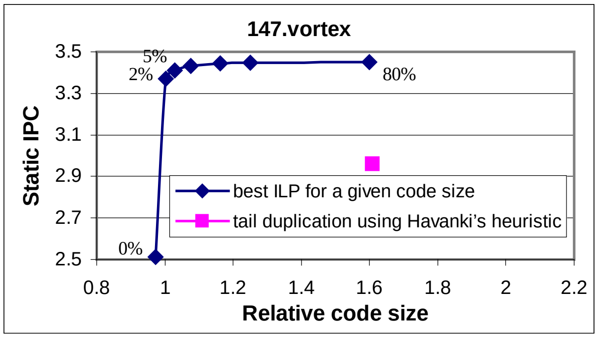

Figure 6. The relationship of ILP vs. code size of benchmark *vortex*

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

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

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

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

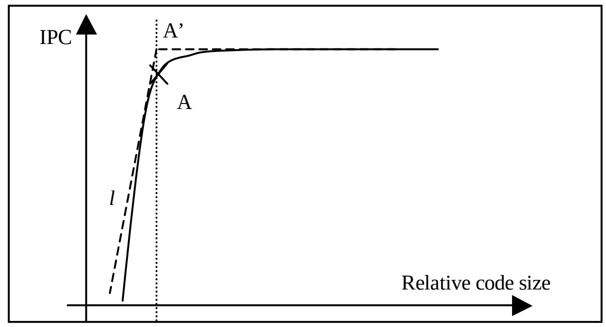

Figure 7. The solution to optimal code size efficiency

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

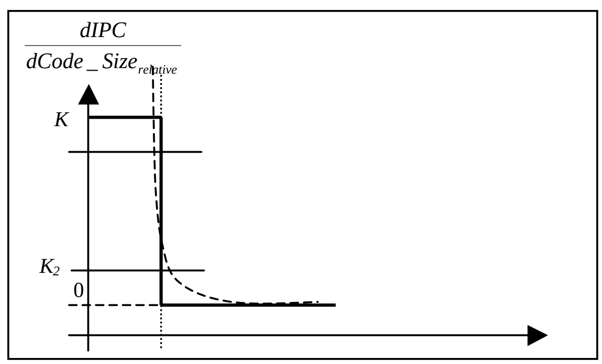

Figure 8. The derivative of the IPC vs. code size curve

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

Table 4. The statistics of operations with different execution frequencies

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

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

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

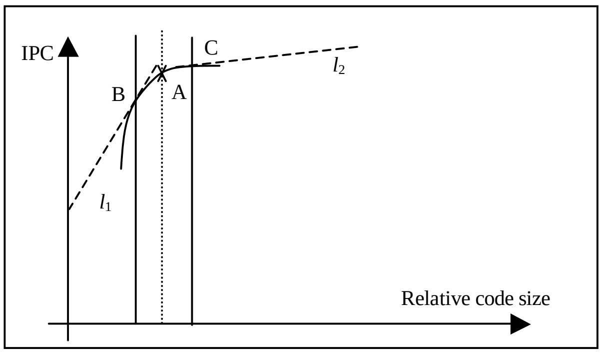

Figure 9. The robustness of the threshold scheme (determined by the slope of the tangent lines at points B and C)

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

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

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

Algorithm for finding the best code efficiency based on tail duplications

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

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

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

Table 5. The experimental results for threshold k = 0.577

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

Table 6. The experimental results for threshold k = 0.268

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

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

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

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

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

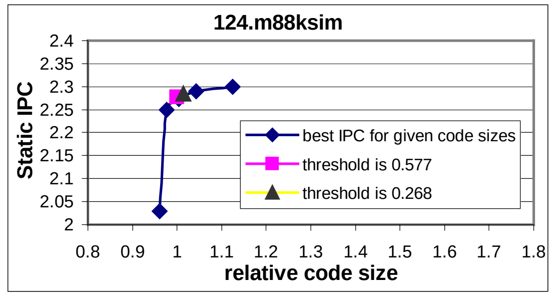

Figure 11. The best code size efficiency found using different thresholds for benchmark *m88ksim*

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

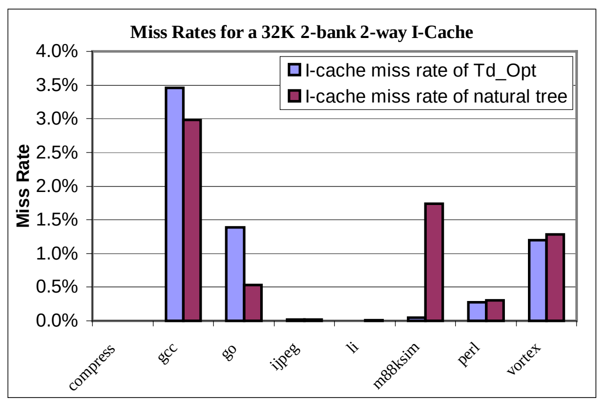

Figure 12. I-cache miss rates for natural treegion and the optimal efficiency results obtained with threshold as 0.577

In Figure 12, benchmarks *gcc* and *go* show significant increases in I-cache miss rate due to the code size increase of the optimal efficiency results while other benchmarks exhibit similar or smaller I-cache miss rates. The reason for the decreases in I-cache miss rates is mainly due to the effect that the tail duplication in optimal efficiency results increases the sequential locality of the frequently executed regions, as observed in [3]. Another fact that improves the I-cache performance is that the tail duplication enables the treegion scheduler to produce a denser schedule of the operations (i.e., more operations in each multi-op). As a result, the number of I-cache accesses is reduced and so is the number of I-cache misses. Figure 13 shows the ratio of I-cache misses of the optimal efficiency results to the natural treegion results. It can be seen from Figure 12 and 13 that although the optimal efficiency results of the benchmark *gcc* has a higher miss rate than natural treegion results, it has smaller I-cache miss penalties due to the reduced number of accesses. In average, the I-cache miss penalties of optimal efficiency results have a 4% decrease comparing to the natural treegion results for a 32KB I-cache.

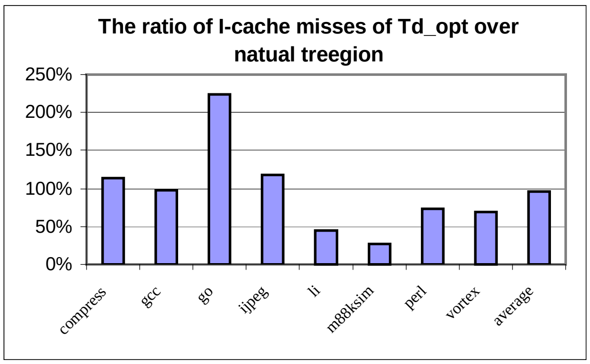

Figure 13. The ratio of I-cache misses of optimal efficiency results over natural treegion results

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

5. Conclusion

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

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

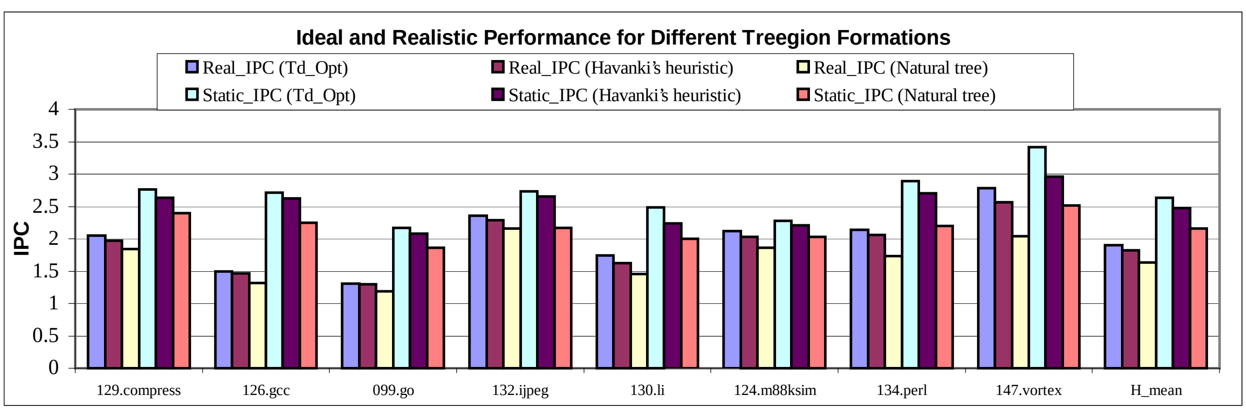

Figure 14. The ideal and realistic performance for different treegion formations

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

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

6. Acknowledgments

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

7. References

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

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

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

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

[5] W.W. Hwu, S.A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. "The Superblock: An effective way for VLIW and superblock compilation." *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

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

[7] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann "Effective compiler support for predicated execution using the Hyperblock" *Proc. 25th Ann. Int'l Symp. Microarchitecture (MICRO25)*, December 1992.

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

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

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

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

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

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

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

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

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

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

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

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

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

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