

Code Size Aware Compilation for Real-Time Applications

Huiyang Zhou

Department of Computer Science

University of Central Florida

zhou@cs.ucf.edu

Abstract

Statically constructed plan of execution (POE) and aggressive instruction level parallelism (ILP) exploitation make EPIC/VLIW processors appropriate for high performance real-time systems. On the one hand, the compiler controlled POE makes the worst-case execution-time (WCET) analysis more accurate as run-time variations are minimized. On the other hand, the compiler can leverage ILP optimizations and instruction scheduling to explicitly reduce the WCET of real-time tasks, which in turn improves the system level schedulability. The ILP optimizations, however, could result in excessive static code size increase if they are not performed carefully, which incurs additional system cost and potential performance degradation due to the affected I-cache performance. In this paper, we propose a low complexity, systematic way to selective perform the code size related ILP optimizations including if-conversion, loop unrolling, and tail duplication, so that the WCET is significantly reduced at the cost of minor static code size increase. We also show that the coupled instruction scheduling algorithm, treegion scheduling, suits the real-time systems well due to its capability to optimize multiple control paths simultaneously.

1. Introduction

Due to the advantage in hardware complexity and power consumption, EPIC/VLIW style processors [25] are widely used in media-based embedded systems. The design philosophy of EPIC/VLIW processors places the compiler in control of the plan of execution (POE) and uses the compiler to exploit the rich instruction level parallelism (ILP) features provided by the hardware to achieve the high performance. Such design philosophy makes EPIC/VLIW processors appropriate for high performance real-time systems. On the one hand, the compiler controls POE so that the worst-case execution-time (WCET) analysis can be more accurate since run-time variations (e.g., due to dynamic

instruction scheduling) are minimized. The hardware features of EPIC/VLIW processor such as static branch prediction and explicit cache level specification also help to reduce the run-time variations due to dynamic branch prediction and caching effects. On the other hand, the compiler can leverage aggressive ILP optimizations and instruction scheduling to explicitly reduce the WCET of real-time tasks, which in turn enhances the system level schedulability. The ILP optimizations, however, could result in excessive static code size increase if they are not performed carefully, which incurs additional system cost and even potential performance degradation due to the affected I-cache performance.

In this paper, we propose a low complexity, systematic approach to selective perform the code size related ILP optimizations including if-conversion, loop unrolling, and tail duplication, so that the WCET is significantly reduced at the cost of minor static code size increase. Our approach is based a notion of code size efficiency of each ILP optimization instance. The code size efficiency is defined as the ratio of the WCET reduction over the static code size increase associated with each individual ILP optimization instance. Since the WCET calculation requires compiled/scheduled code, we propose a set of performance bounds to evaluate the WCET reduction. These performance bounds have low computational complexity and yet capture instruction-scheduling effects accurately so that the computationally expensive instruction scheduling is not necessary. Then, a general framework is developed to selectively perform the ILP optimizations based their code size efficiencies. The experimental results show that our approach achieves remarkable WCET reductions and increases the static code size slightly for most workloads.

The remainder of the paper is organized as follows. Section 2 discusses background work. Using performance bounds to evaluate the WCET reduction is contained in Section 3. Section 4 presents the algorithm to selectively perform ILP optimizations. The experimental methodology and results are in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2. Background

2.1. Real-time systems scheduling and WCET analysis

In real-time systems, a task needs to satisfy both functional and temporal requirements to achieve overall correctness [7][16]. The functional requirements are defined based on program semantics to generate correct outputs from inputs and the temporal requirements define the upper bounds (or deadlines) and lower bounds in time for such input-output transformation. A real time system may have many such tasks (periodic or sporadic) and they are scheduled (called task scheduling) to meet the overall timing requirements of the system.

In order to guarantee a task to be finished by a specified deadline, the worst-case execution-time (WCET) analysis, either statically or dynamically, is commonly used. Task scheduling then sets different priorities for different tasks accordingly. Due to its evident impact, task scheduling for real-time applications is an active research topic. In this thesis, we look at the problem from a different point of view. Instead of focusing on task scheduling algorithms, we focus on *intra-task instruction level scheduling*; more specifically we explore ILP optimizations and instruction scheduling to reduce the WCET of each task. The reduction in WCET is important since (1) it can reduce the WCET of a task to make it meet its deadline specification; (2) it enables the processor to serve more tasks, thereby achieving better utilization; (3) the system can run at a lower frequency/voltage to save power/energy when enough slacks can be produced.

Most of previous work on real-time scheduling takes the compiled task programs as input and perform either static or dynamic timing analysis to determine the WCET. Little work is done at instruction level (optimization or scheduling) to reduce the WCET. Gerber and Hong [7] proposed a scheduling approach called *structural code motion*. First, the task-level timing requirements are broken down into the event level. A new language based on the C language, called the Time-Constrained

Event Language (TCEL), is developed to express detailed event-based timing constraints. Then, the code is partitioned into sections based on observable events. Trace-based scheduling [6] is used to schedule each section. The critical traces (i.e., the traces whose execution time is larger than the timing constraints) are examined and code motion may be performed to move operations across sections to make the WCET relationship between observable events meet the timing constraints. Such code motion can be unconditional (or safe) or control speculative. Since trace scheduling focuses on one trace at a time, such code motion results in considerable bookkeeping codes and could potentially increase the criticality of other traces. As a result, the critical paths are repetitively checked and scheduled. Compared to this approach, our approach uses treeregion based scheduling [10], which enables speculation from multiple control paths simultaneously and limits the enumeration of critical paths, and explores various ILP optimizations to reduce WCET. Another technique to increase task schedulability is based on the concept of imprecise computation [8][17]. If the purpose of some computation is known statically as refining the results, such computation can be skipped without affecting system sustainability, i.e., the quality of computation is traded for the timeliness of the results.

In [15][27], algorithms have been proposed to schedule instructions with timing constraints (release times and deadlines) on ILP processors. These algorithms are targeted toward single-issue pipelined processors and work on the basic-block (BB) level. The proposed algorithms guarantee to find a feasible schedule for a range of special cases.

2.2. Treeregion scheduling for EPIC/VLIW processors

Treeregion scheduling [10][29] is an aggressive global instruction-scheduling algorithm proposed for wide issue EPIC/VLIW processors. It has two steps: treeregion formation and tree traversal scheduling (TTS). A treeregion is a single-entry/multiple-exit nonlinear code region that consists of basic blocks

(BBs) with the control-flow forming a tree, as illustrated in Figure 1. Based on the control flow graph (CFG) in Figure 1a, two treeregions are formed. The treeregions that are formed without any tail duplication are referred to as *natural treeregions*. When the tail duplication [3][22] is applied, a larger treeregion 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 treeregion is formed containing all the BBs in the CFG, as shown in Figure 1b. Such duplication enables the speculation from BB7, BB8, BB9 and their duplicates, thereby increasing ILP. The trade-off for exposing ILP through treeregion formation is the code-expansion resulting from the duplicates of BB7, BB8 and BB9.

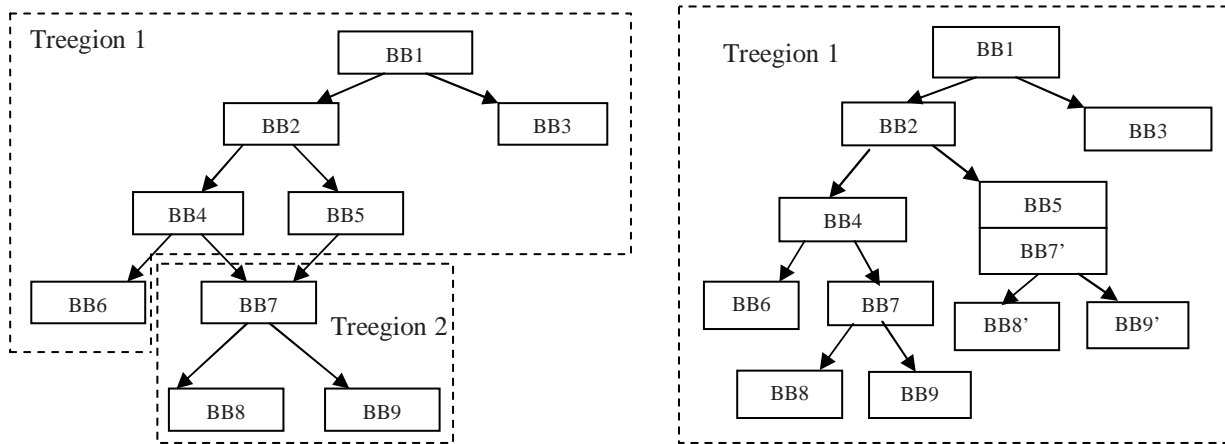


Figure 1. (a) The CFG and the treeregions constructed; (b) The treeregion constructed after the tail duplication.

During the tree traversal scheduling (TTS), the BBs in a treeregion are scheduled in a predetermined traversal order based on the treeregion 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. By this way, it enables the control speculation¹ from all the paths starting from the BB. The candidate instructions are scheduled based on an order determined by a heuristic that includes their execution frequency, exit count, and data dependence height. The details

¹ In this paper, the data speculation is not considered as it involves recovery code and could potentially increase WCET. For control speculation, such recovery is not necessary as real-time programs are well behaved and would not throw an exception in normal execution. So, we assume the deferred report of an exception [19][13] is enough when an exception really happens.

of tree traversal scheduling can be found in [29]. In this paper, we modify the original TTS algorithm to be *profile independent* since the objective here is to optimize the worst-case scenarios instead of optimizing the most frequently executed paths.

3. Using Performance Bounds to Evaluate the WCET Reduction

3.1. Performance bounds at instruction scheduling region level

Actual WCET computation requires the scheduled code, which implies that the computationally expensive instruction scheduling needs to be performed. Since we are interested in the WCET reduction resulting from ILP optimizations instead of the actual WCET, we propose to use performance bounds based on the *un-scheduled* code to estimate the WCET. The reduction in performance bounds then reflects the performance potential in the WCET reduction. In order to capture instruction-scheduling effects accurately, we define such performance bounds at the granularity of the instruction-scheduling region level. It is important since the instructions are not moved across the scheduling region boundary but are moved inside the scheduling region during the instruction-scheduling phase.

Many instruction scheduling algorithms such as superblock scheduling [12], hyperblock scheduling [20], and treeregion scheduling use a single-entry multiple-exit code region as a basic scheduling region. We define the lower bound of worst-case execution time (LBWT) of such a region as Equation 1:

$$\begin{aligned}
 LBWT &= \underset{path_i}{Max} LBET_{path_i} \\
 &= \underset{path_i}{Max} \left[\underset{path_i}{Max} (data_dependence_bound_{path_i}, resource_bound_{path_i}) \right]
 \end{aligned} \tag{1}$$

As shown in Equation 1, the lower bound of WCET (LBWT) for a multi-path region is simply the maximum of lower bound execution time (LBET) of each path, which is 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 data dependence bound assuming software renaming is

available at schedule time to remove false register dependencies. By calculating the data dependence bound along each path, the potential control speculation effect is captured implicitly as control dependence is not enforced. Also, using only the true data dependence simplifies the bound computation as it is an $O(N)$ computation, where N is the number of instructions along the control path.

Resource bound in Equation 1 is calculated similar to the ResMII calculation in iterative modulo scheduling [24], as follows.

$$resource_bound_{path_i} = \text{Max}_k \lceil (Num_Insn_k / Num_FU_k) \rceil \quad (2)$$

In Equation 2, Num_Insn_k represents the number of operations that use the function unit type k . Num_FU_k represents the number of the function units of type k available in the processor. The ratio (ceil) of these two numbers shows the resource constraints of the function units of type k . Then, resource bound is calculated as the maximum constraint among all types of function units.

Note that in Equation 1, the LBWT of a multi-path region means that the worst case of control flow (i.e., the path with longest execution time) is assumed while the lower bound execution time (LBET) is used for each path. Since such a lower bound is used, the actual execution time along the longest path could potentially exceed this lower bound. So, it apparently conflicts the purpose of the WCET. However, remember that we use the changes in LBWT to measure the impact of WCET reduction due to code optimizations instead of using LBWT directly as the final WCET measure. Using LBET eliminates the computationally expensive instruction scheduling and yet provides an accurate estimate of the actual execution time. Moreover, this LBWT can be used to check the soundness of the deadline setting: if the predetermined deadline exceeds the LBWT, it is impossible that the task can be finished in time when the longest control path is taken. In such a case, the system has to reassign the deadlines, adopt a more powerful processor, or optimize the code more aggressively.

3.2. Performance bounds at function and program level

Computing *LBWT at function level* is complicated due to complex CFG and multiple regions in a function body. Here, we use a similar approach to the static WCET analysis for scheduled code, in which the analyzer derives the WCET for each path, then for loop bodies, and finally for functions in the program. The WCET of the *main* function is simply the WCET for the entire program. Compared to this path-based static analysis approach for scheduled code, treeregion-based LBWT provides an efficient way to capture the treeregion scheduling effect accurately and limits the numeration of the possible control paths. Next, we use an example to derive this treeregion-base LBWT analysis (though the same derivation applies to any single-entry multiple-exit type of scheduling region). We start with an innermost loop body, which may contain more than one treeregion. One such example is shown in Figure 2.

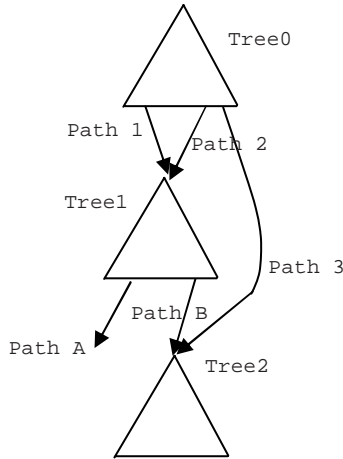


Figure 2. The LBWT for the innermost loop body.

The CFG in Figure 2 represents an innermost loop body containing three treeregions. To account for the control dependencies among these treeregions, we extend Equation 1 to Equation 3 to compute LBWT for each treeregion. The LBWT for treeregion 0 in this example is then the LBWT for the innermost loop body.

$$LBWT = \text{Max}(LBET_{path_1} + LBWT_{base_path_1}, \dots, LBET_{path_k} + LBWT_{base_path_k}) \quad (3)$$

In Equation 3, LBWT of a treeregion is computed as the maximum of LBWT of every path in the treeregion, which is turn defined as the sum of LBET of the path ($LBET_{path_i}$) and LBWT of the treeregion that the path leads to ($LBWT_{base_path_i}$). The term $LBET_{path_i}$ is defined as before, i.e., the maximum of the data dependence bound and the resource bound. The term $LBWT_{base_path_i}$ is computed recursively using the same Equation 3. For exit paths or return paths, such $LBWT_{base}$ is zero. For the code example in Figure 2, the overall LBWT (i.e., LBWT of treeregion 0) is computed as follows:

$$LBWT_{treeregion0} = \text{Max}(LBET_{path_1} + LBWT_{base_path_1}, \dots, LBET_{path_k} + LBWT_{base_path_k}) \\ = \text{Max}(LBET_{path_1} + LBWT_{treeregion1}, LBET_{path_2} + LBWT_{treeregion1}, LBET_{path_3} + LBWT_{treeregion2}).$$

LBWTs of treeregion 1 and treeregion 2 can be computed in turn as:

$$LBWT_{treeregion1} = \text{Max}(LBET_{path_A}, LBET_{path_B} + LBWT_{treeregion2}); \\ LBWT_{treeregion2} = \text{Max}(LBET_{paths_in_treeregion2}).$$

For an outer loop body or a CFG containing loop structures, such as the CFG shown in Figure 3, the LBWT can be computed as follows,

$$LBWT_{treeregion0} = \text{Max}(LBET_{path_1} + LBWT_{loop_A}, LBET_{path_2} + LBWT_{treeregion1}).$$

where LBWT of the loop A is computed as $LBWT_{loop_body_A} * loop_count_A + LBWT_{treeregion1}$. LBWT for the loop body ($LBWT_{loop_body_A}$) can be computed using Equation 3 if it contains more than one treeregion and the loop count is determined from the workload specification or from profiling.

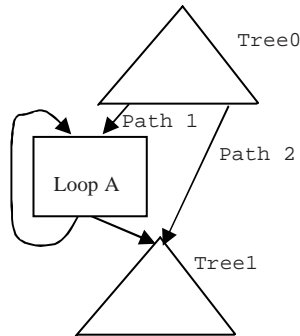


Figure 3. The LBWT for outer loops or code regions containing inner loops.

LBWT at program level can be computed from the functional level LBWTs by traversing the function call graph (leaf node first). The LBWT of the ‘*main*’ function represents the LBWT of the entire program.

As a final note, if we replace LBET along each path with the actual schedule length/execution time, the LBWT becomes the WCET.

3.3. An Example

Here, we use a code example to illustrate the LBWT computation and show that treeregion scheduling will result in smaller LBWT compared to superblock scheduling or trace scheduling, making treeregion scheduling more suitable for real-time applications. The code example is a simple diamond structure as shown in Figure 4. The machine model used in this paper has the following configuration: 6-wide issue (2 ALU, 2ALU/LD/ST, 2 BR, e.g., Itanium-I or II [13]); load operations having a 2-cycle latency and all other integer operations having a 1-cycle latency (except CMP instructions which can be issued at the same cycle as the consuming branch).

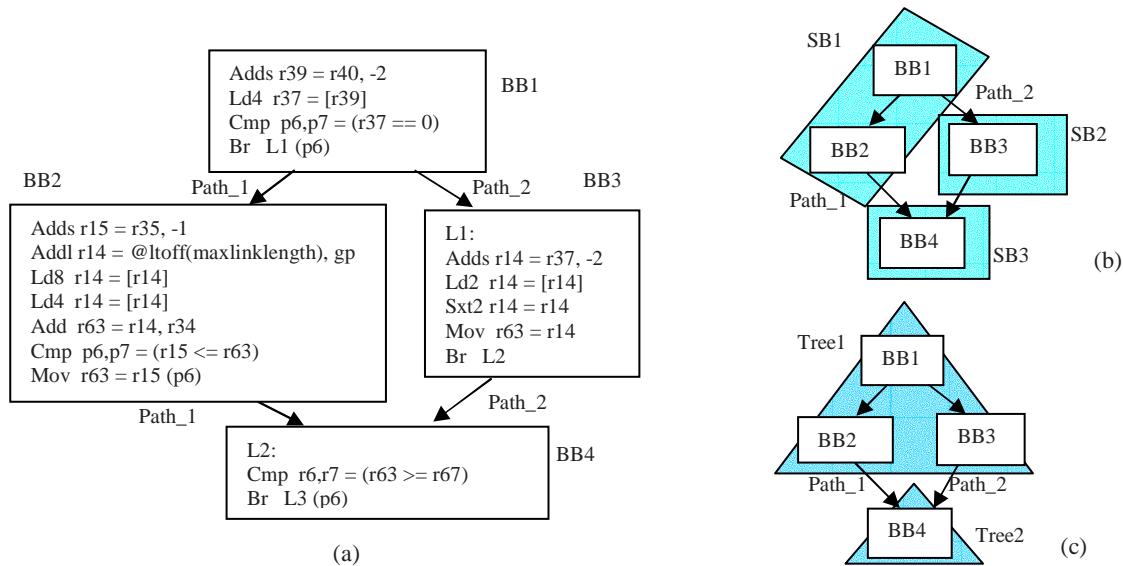


Figure 4. (a) A code example for the LBWT computation; (b) superblocks formed without tail duplication; (c) treeregions formed without tail duplication (natural treeregions).

First, we compute superblock-based LBWT of the code segment (Figure 4b), which is the same as $LBWT_{SB1}$. Using Equation 3, it can be computed as:

$$\begin{aligned} LBWT_{SB1} &= \text{Max}(LBET_{path_1}+LBWT_{SB3}, LBET_{path_2}+LBWT_{SB2}) \\ &= \text{Max}(LBET_{path_1}+LBWT_{SB3}, LBET_{path_2}+ LBET_{SB2_path} +LBWT_{SB3}) \\ &= \text{Max}(8 + 1, 4 + 5 + 1) = 10 \text{ cycles.} \end{aligned}$$

From the computation, it seems that control path 2 forms the critical path.

Using treeregions as basic scheduling regions (Figure 4c), the LBWT of the code segment is $LBWT_{tree1}$, which is computed as follows:

$$LBWT_{tree1} = \text{Max}(LBET_{path_1}+LBWT_{tree2}, LBET_{path_2}+LBWT_{tree2}) = \text{Max}(8 + 1, 5 + 1) = 9 \text{ cycles.}$$

Now the critical path is due to control path 1. Compared to superblock-based LBWT, treeregion based LBWT incorporates the control speculation from BB3 along path 2. So, the execution time along path 2 is reduced to 6 cycles and overall LBWT is reduced to 9 cycles. This illustrates that treeregion scheduling is an appropriate scheduling framework for real-time applications due to its capability to perform speculation from multiple control paths simultaneously. In this example, we do not consider the branch prediction effect on WCET or LBWT for conciseness. We discuss it in Section 4 when if-conversions [1][23] are selectively performed.

4. Reduce the WCET Using ILP Optimizations

4.1. Code size efficiency for ILP optimizations

As the code size related ILP optimizations trade static code size increase for reduced WCET, one direct measure of the effectiveness of such an optimization is to use the ratio of the WCET reduction over the code size increase. As discussed in Section 3, the actual WCET calculation requires time-consuming instruction scheduling and the LBWT reduction is used instead to evaluate the effectiveness of code optimizations. So, we define the *instantaneous code size efficiency* for each individual optimization instance as Equation 4.

$$Efficiency_{inst} = \frac{LBWT_{before_individual_application} - LBWT_{after_individual_application}}{code_size_{after_individual_application} - code_size_{before_individual_application}} \quad (4)$$

The numerator in Equation 4 represents the LBWT reduction resulting from an instance of a code optimization and the denominator represents the cost of such an optimization in terms of static code size increase. Using loop unrolling as an example, if we unroll a particular loop once, the instantaneous efficiency of such an unrolling is the performance gain in LBWT reduction divided by the size of the loop body. Since there could be many loops in a program, there is one such instantaneous efficiency associated with each of them.

Next, we use an example to illustrate the code size efficiency computation and illustrate the effect of the ILP optimizations to reduce LBWT (and WCET eventually). The code segment, as shown in Figure 5, is part of the audio encoding/decoding kernel extracted from the benchmark *adpcm* in the MiBench suite [9]. The small data input set is used for this benchmark.

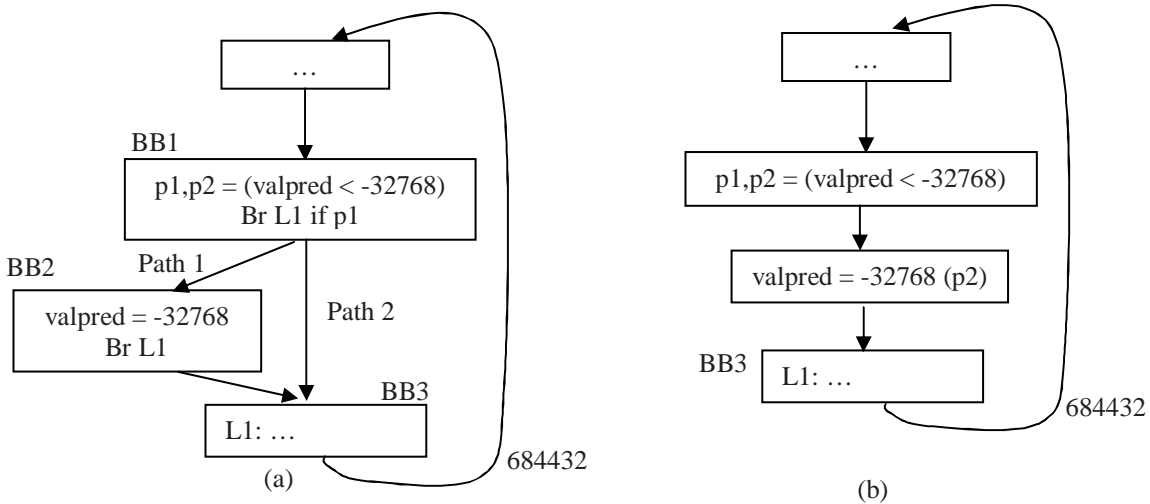


Figure 5. (a) The code segment from the benchmark *adpcm* (compiled using *gcc 3.1 -O 1*); (b) the code after the if-conversion.

To account for the branch misprediction penalty associated with the conditional branch in the code segment, the static branch prediction, which favors the longer path, is used [1]. Then, the misprediction penalty is added to the shorter path as part of its execution time. For loop back branches, either backward-taken/forward-not-taken is used as the static prediction or some loop count hardware

features [13] are used to achieve perfect prediction. In this paper, we assume that the loop back branch can always be predicted correctly and no penalty is associated.

Using the approach presented in Section 3.2, we compute the LBWT for this benchmark as 24,012,356 cycles assuming a 10-cycle branch misprediction penalty. The ILP optimization, if-conversion can remove the conditional branch in Figure 5a and result in the straight-line code shown in Figure 5b. In this case, the LBWT is reduced by 26% to 17,852,468 ($= 24,012,356 - 9 * 684432$) cycles. Considering the code size changes due to this if-conversion instance, we can see there is a 2-instruction reduction (the conditional branch in BB1 and the unconditional branch in BB2). So, the code size efficiency of this particular instance of if-conversion is -8,926,234 cycles/insn ($= 17,852,468 / -2$). This negative number represents a highly desired extreme of the code optimization as it reduces both the LBWT/WCET and static code size.

We can further optimize the code segment in Figure 5 using tail duplication and loop unrolling. Duplicating BB3 enables control speculation for instruction in BB3 and its duplicate, resulting in another 2,631,720 cycle reduction in LBWT and 24 duplicated instructions. The code size efficiency of duplicating BB3 is then 109,655 cycles/insn ($= 2,631,720 / 24$) meaning that each additional instruction introduced by this tail duplication instance leads a 109,655-cycle LBWT reduction. Unroll this loop once will reduce the LBWT further by 482,016 cycles while introduces 83 replicated instructions. So, its instantaneous code size efficiency is 5,807 ($= 482,016 / 83$) cycles/insn.

From this example, we can see that ILP optimizations can reduce LBWT significantly but different optimization instances have different performance impact and the code size increase. Instead of reiterating the effectiveness of ILP optimizations, our objective is form a general framework to selectively perform such optimization instances so that the LBWT/WCET reduction is achieved at the minimum cost of code size increase.

4.2. The algorithm to selectively perform ILP optimizations

Based on the quantitative measures of the code size efficiency of ILP optimizations in Section 4.1, we develop an algorithm shown in Figure 6 to selectively perform the optimization.

Algorithm for regulating code size related optimizations in real-time applications

0. Form basic scheduling regions to facilitate LBWT computation and to identify program structures that are candidates for optimizations.
1. Perform code size reducing optimizations: *if-conversion (or predication)*
 - a. For a diamond/hammock structure, compute performance gains of if-conversion.
 - b. If the if-conversion produces positive (or zero) LBWT reduction, perform it
 - c. If the performed if-conversion results in a new diamond/hammock for its parent branch, continue to check this parent branch for if-conversion.
 - d. Repeat step 1a – 1c, until no more diamond/hammock structures need to be checked.
2. Perform code size increasing optimizations: *loop unrolling and tail duplication*
 - a. Compute instantaneous code size efficiency for each loop unrolling / tail duplication candidate using Equation 4.
 - b. Search the candidate list to find the one with the highest efficiency.
 - c. If the selected candidate passes the feasibility check, perform the optimization and update the efficiency of candidates affected by the optimization. (The feasibility check may include local/global code size constraints, register pressure, etc.)
 - d. Repeat step 2a – 2c, until the overall code size reaches a limit or there are no more candidates.

Figure 6. The algorithm to selectively perform ILP optimizations.

This algorithm has three steps. In Step 0, the preparation step, the basic scheduling regions are formed (e.g., natural treeregions) and the optimization candidates are identified. Next, the optimization candidates are treated differently based on their instantaneous code size efficiency characteristics in Step 1 and Step 2. Optimizations with positive LBWT reduction and negative code size increase are examined first in Step 1. Then, an iterative approach is used to selectively perform code-expanding optimizations, as shown in Step 2 of the algorithm. First, step 2a computes the efficiency of all potential optimization instances. Then, the best candidate is selected 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. 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. The feasibility

check in this step basically makes sure a particular optimization instance will not result in excessive resource utilization, e.g., the size of a loop body is less than level one I-cache size.

5. Results

5.1. Experimental methodology

In our experiments, we use selected benchmarks from both MiBench benchmark suite [9] and SPEC2000 INT benchmark suite [11] to evaluate the proposed algorithms. As our objective is to reduce the WCET, we excluded the benchmarks containing recursive function calls or un-structural loops since they present obstacles for our current LBWT/WCET analysis. The selected benchmarks are first compiled into IA-64 assembly using the *gcc* compiler (version 3.1). As our focus is ILP optimizations, we use level one optimization of *gcc* to perform classical optimizations. The resulting IA-64 assembly codes are then parsed into the LEGO compiler framework [14], which we use to implement the algorithms in this paper. For the workloads from MiBench, ‘small’ data input sets are used to determine the loop counts. For the benchmark *bzip2*, we use reference input data set and skip the first 500 million instructions and profile the next 500 million instructions. In the experiments, after the ILP optimization phase, the code is scheduled using treeregion scheduling and the WCET is computed.

5.2. Regulating the code size reducing optimizations: if-conversion

Step 1 of the algorithm shown in Figure 6 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 4.1, we use static branch prediction to estimate branch misprediction impact on LBWT/WCET assuming that each misprediction associates a 10-cycle penalty.

As stated in Section 5.1, the *gcc* optimization level one is used to generate the IA64 assembly. However, the level one optimization of *gcc* also produces predicated instructions (i.e., the level one optimization performs if-conversion). So, in the first experiment, we modified the *gcc* 3.1’s source code to turn-off its if-conversion transformation and use the algorithm in Figure 6 (step 1) to perform if-conversion. The results in WCET reduction are shown in Figure 7. From Figure 7, it can be seen that aggressive if-conversion can reduce WCET significantly, up to 80% for the benchmark *adpcm* since its source code contains many ‘if-then’ and ‘if-then-else’ structures. In the benchmarks *rijndael* and *sha* (both are security benchmarks), the encryption/decryption kernel contains mostly straight-line instructions. Therefore, if-conversion yields negligible impacts on the WCET reduction.

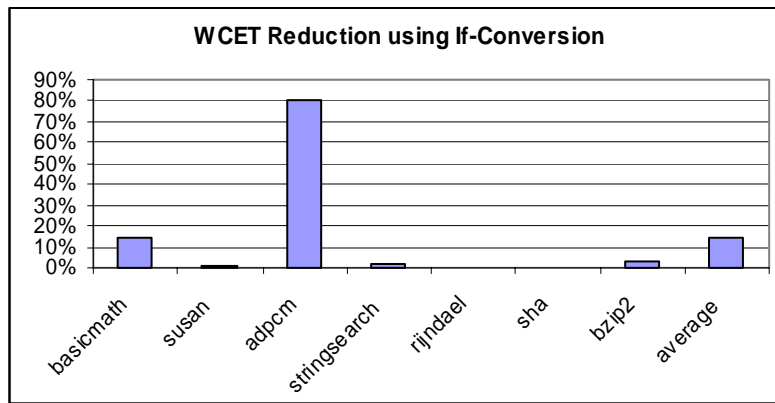


Figure 7. The WCET reduction using if-conversion.

In the next experiment, we turn on the *gcc*’s if-conversion option to generate the assembly and then use our algorithm to perform if-conversion. The results shown in Table 1 indicate that our algorithm can improve upon *gcc*’s if-conversion algorithm.

Table 1. The results of if-conversion based on *gcc*’s predicated code.

	basicmath	susan	adpcm	stringsearch	rijndael	sha	bzip2
If-conversions (by <i>gcc</i>)	3	72	21	11	13	3	113
Number of conditional br	18	325	8	50	56	14	487
If-conversions with pos. gain	0	30	2	0	0	0	23
If-conversions with zero gain	0	18	0	0	1	0	2
If-conversions with neg. gain	0	0	0	0	0	0	0

Interesting observations can be made from Table 1. The first row in Table 1 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 not appropriate for if-conversion either due to the complex CFG, which inhibits diamond/hammock detection, or the detected diamond/hammock containing function calls, returns, or indirect branches (we excluded the case that both paths contain the same function calls or returns). In such cases, if-conversion may hurt the WCET since if-conversion introduces more conditional function calls and returns, which in turn incurs more branch misprediction penalties. For those if-convertible branches, our algorithm computes the LBWT reduction. Using the benchmark *adpcm* as an example, *gcc* converts 21 conditional branches and there remain 8 conditional branches in the program. Our algorithm finds that 4 of them cannot form a diamond/hammock structure. For those that can form a diamond/hammock, 2 of them have either a function call or a return instruction along the paths. For the remaining 2 conditional branches, both of them produce positive LBWT reduction.

Next, we analyze the performance impact of our if-conversions. In this experiment, we perform only the if-conversions that produce positive LBWT reductions. Although the number of these if-conversions seems limited (2 to 30, the third row in Table 1), additional WCET reduction (up to 26%) can be achieved, as shown in Figure 8.

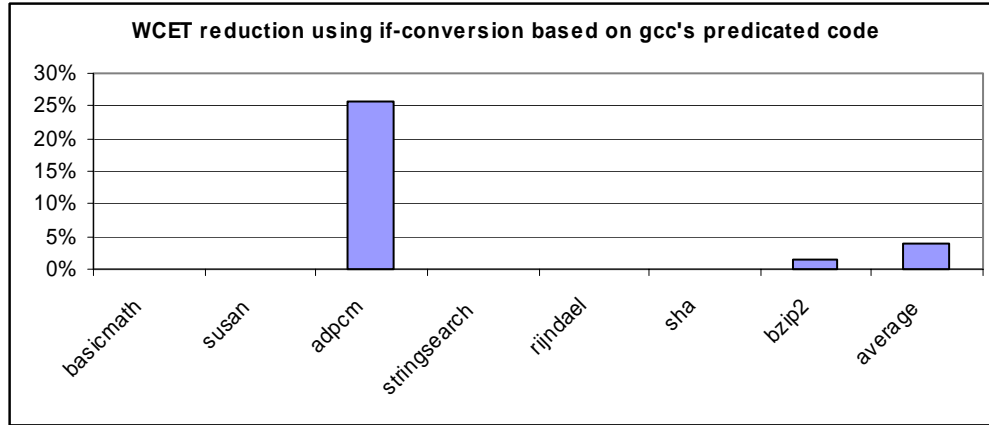


Figure 8. The WCET reduction achieved by if-conversion based on predicated code by *gcc*.

Finally, we analyze the code size reduction impact of if-conversions. We choose to perform if-conversion with positive or zero LBWT reduction in this experiment. Assuming each conversion saves two instructions in IA-64, the overall code size reduction is computed and is shown in Figure 9. Remember that this reduction is achieved on the code that has already been predicated by *gcc*. This shows that our algorithm performs if-conversion more aggressively in reducing the code size. From Figure 9, it can be seen that if-conversion reduces static code size up to 1.85% (the benchmark *adpcm*) and 0.60% on average. Although these numbers seem to be trivial, in the next subsection, we will show that utilizing such a small amount of code size can lead to large WCET reduction.

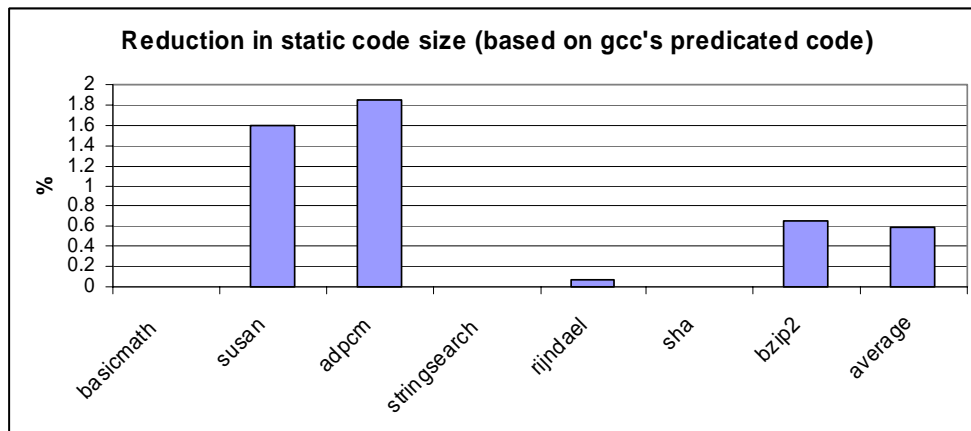


Figure 9. The code size reduction from if-conversion based on predicated code by *gcc*.

5.3. Regulating code size increasing ILP optimizations: loop unrolling and tail duplication

Step 2 of the algorithm shown in Figure 6 regulates code size increasing optimizations (tail duplications and loop unrolling). It iteratively selects and performs one instance of tail duplication or loop unrolling with the highest instantaneous code size 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 5%, 10%, and 20% of its original size (i.e., the optimization stops when the overall code size increase reaches the limit). The corresponding WCET reductions are shown in Figure 10.

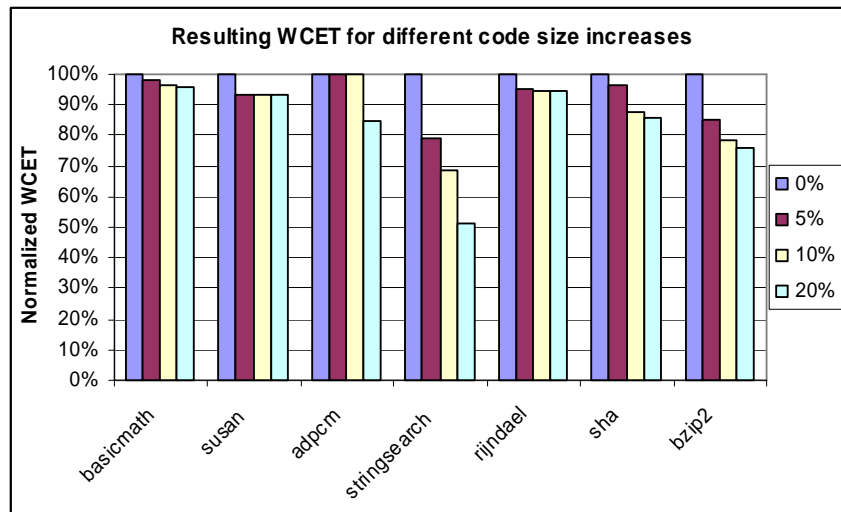


Figure 10. The WCET reduction for different code size increase using code size enlarging optimizations.

From Figure 10, it can be seen that the significant reductions in WCET is achieved from code size enlarging optimizations. The benchmark *stringsearch* shows the largest WCET reduction. Looking into its optimization process, we found that the gain is mainly from unrolling two frequently executed loops. These loops have no loop-carried dependence, thus producing large performance improvement from unrolling until the resource bound becomes the bottleneck. The benchmark *basicmath*, on the other hand, shows the smallest WCET reduction. This benchmark contains several basic math functions, including *SolveCubic*, *usqrt*, and *rad2deg*. The most frequently called function *SolveCubic* contains only one treeregion; therefore it cannot be optimized further with either tail duplication or loop

unrolling. As a result, this benchmark does not show big WCET reduction though other functions are quite optimized with unrolling and tail duplication. Also, from Figure 10, it can be noticed that for benchmark *adpcm*, there is no reduction when code size increase limit is set to 5% or 10% while it shows 18% reduction when the limit is 20%. The reason is that in this benchmark, the main loop body of the encoding contains a hammock. Duplicating the merge block of this hammock has large performance gains but the size of this block is about 11% of its original size.

The *diminishing returns* phenomenon can also be observed from Figure 10. Using the benchmark *bzip2* as an example, the first 5% code size increase leads to a 15% WCET reduction while the next 5% (total 10%) code size increase leads to an additional 7% reduction and another 10% (total 20%) code size increase results in another 2% WCET reduction. This shows that during the iteration process of performing loop unrolling and tail duplication, the efficiencies of the initial picks are much higher than remaining ones. Other benchmarks share the similar trend and most of the WCET reduction is achieved in the first 5%-10% code size increases. For the benchmark *stringsearch*, there exists significant WCET reduction until the code size increase reaches 40%, as seen in Figure 11.

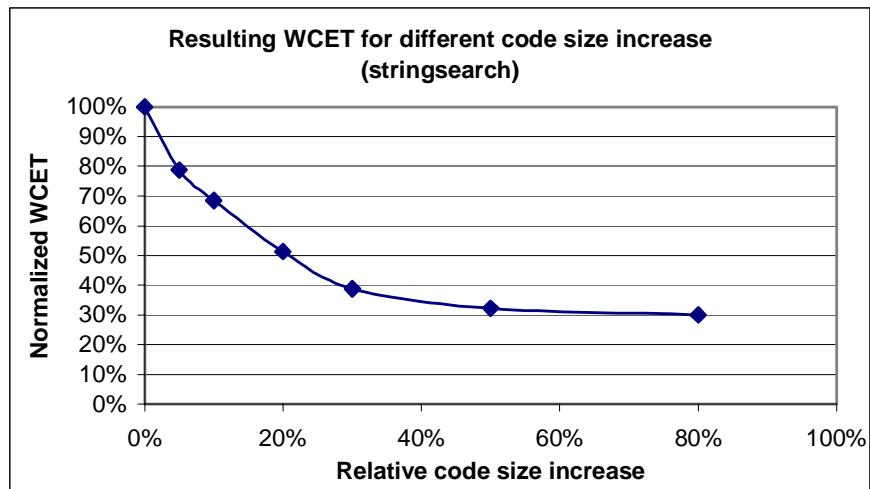


Figure 11. The diminishing returns exhibited in the benchmark *stringsearch*.

There are two fundamental reasons for such diminishing returns phenomenon as observed for general purpose computing [28]. First, based on the definition of code size efficiency, the effects of

optimizations (LBWT/WCET reduction) happened inside loop bodies are amplified by the factor of the loop count. The well-known ‘90/10 rule’ points out that a small part of the static code (hot portions) consumes over most of the execution time in general purpose computing. The similar rule also holds for real-time applications as most of the WCET is spent on some heavily executed loops and functions called from such loops. After performing optimizations in these ‘hot’ portions of code, the remaining optimizations should have much lower efficiencies. Secondly, high efficiency also requires that the resulting code must have significantly reduced LBWT, i.e., the optimization instance must reduce the DDG height without causing any resource conflict problem. This requirement filters the optimizations happening in hot portions of a program.

One practical implication of this diminishing return phenomenon is that we can monitor the code size efficiency during the iteration process. If the highest one is less than a threshold, we can expect that further optimization will have minor impact on WCET reduction while involving large static code size increase.

6. Summary and Future Work

In this paper, we advocate using compile-time ILP optimizations and instruction scheduling to reduce the WCET. We focus on regulating three code size related ILP optimizations, if-conversion, tail duplication, and loop unrolling, to reduce the WCET while limiting the associated code size increase at the same time. We propose to use performance bounds to evaluate the performance potential of ILP optimizations so as to avoid the computationally expensive instruction scheduling. Then, we develop a general framework to selectively perform the ILP optimizations based on their performance potential and the cost in code size increase. The experimental results show that by combining aggressive instruction scheduling and carefully performed ILP optimizations the WCET can be significantly reduced at cost of minor static code size increase.

Our work can be extended in several ways. First, we can integrate more advanced WCET analysis approaches [4][5][18][21][26] to account for the branch prediction and cache effects more accurately. The instruction cache effects are particularly important if the instruction cache size cannot hold the hot spots of the task. This paper addresses this issue indirectly by setting a local/global limit on static code size increase. Accurate analysis of the cache behavior would help to derive such a limit. Secondly, the general framework to selectively perform ILP optimizations can be extended to include more optimizations such as function inlining in addition to the three ILP optimizations considered in this paper.

7. References

- [1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence", *Proceeding of 10th ACM Symposium on Principles of Programming Languages*, 1983.
- [2] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems". *Proc. of the 30th Int'l Symp. on Computer Architecture (ISCA-30)*, 2003.
- [3] D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling", *Proc. 24th Ann. Int'l Symp. Microarchitecture (MICRO24)*, 1991.
- [4] K. Chen, S. Malik, and D. August, "Retargetable static timing analysis for embedded software", *Int'l Symp. on System Synthesis (ISSS'01)*, 2001
- [5] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction", *International Journal of Time-Critical Computing Systems*, 2000.
- [6] J. A. Fisher, "Trace Scheduling: A technique for global microcode compaction", *IEEE Trans. Computer*, July 1981.
- [7] R. Gerber and S. Hong, "Compiling real-time programs with timing constraint refinement and structural code motion", *IEEE Trans. on Software Engineering*, Vol. 21, No. 5, May 1995.
- [8] P. Gopinath and R. Gupta, "Applying compiler techniques to scheduling in real-time systems", *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS)*, 1990.
- [9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, "MiBench: a free, commercially representative embedded benchmark suite", *2001 IEEE Int'l Workshop on Workload Characterization (WWC-4)*, 2001.
- [10] 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.
- [11] J. Henning, "SPEC2000: measuring CPU performance in the new millennium", *IEEE Computer*, July 2000.
- [12] 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.
- [13] Intel Corp, IA-64 Application Developer's Architecture Guide, 2000.
- [14] The LEGO Compiler. Available for download at <http://www.tinker.ncsu.edu/LEGO>.
- [15] A. Leung, K. Palem, and A. Pnueli, "A fast algorithm for scheduling time-constrained instructions on processors with ILP", *Proc. Of the 1998 Conf. On Parallel Architectures and Compilation Techniques (PACT'98)*, 1998.
- [16] J. W. S. Liu, *Real-Time Systems*, Prentice Hall, Upper Saddle River, New Jersey, 2000.
- [17] J. W. S. Liu, K. J. Lin, C. L. Liu, and C.W. Gear, "Research on Imprecise Computations in Project Quartz", *Proceedings of the 1989 Workshop on Operating Systems for Mission Critical Computing*, 1989.
- [18] T. Lundqvist and P. Stenstrom, "An integrated path and timing analysis method based on cycle-level symbolic execution", *Real-Time Systems*, 17(2/3): 183-207, 1999.
- [19] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of branches", PhD thesis, ECE Department, Univ. of Illinois at Urbana-Champaign, 1996.
- [20] 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.
- [21] F. Mueller, "Timing Analysis for Instruction Caches", *Real-Time Systems Journal*, Vol. 18, May 2000.

- [22] F. Mueller and D. B. Whalley, "Avoiding Conditional Branches via Code Replication", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-1995)*, June 1995.
- [23] J. C. Park and M. S. Schlansker, "On predicated execution", Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, 1991.
- [24] B. R. Rau, "Iterative Module Scheduling", Tech. Rep. HPL-94-115, Hewlett-Packard Laboratories, 1995.
- [25] 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.
- [26] R. White, F. Mueller, C. Healy, D. Whalley and M. Harmon , "Timing Analysis for Data Caches and Set-Associative Caches", *Real-Time Technology and Applications Symposium*, 1997.
- [27] H. Wu and J. Joxan, "An efficient algorithm for scheduling instructions with deadline constraints on ILP processors", *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, 2001.
- [28] H. Zhou, "Using performance bounds to guide code compilation and processor design", *Ph.D. thesis, ECE Department, N.C. State University*, 2003.
- [29] H. Zhou, M. Jennings, and T. M. Conte. "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors". *Proc. of the 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, LNCS, Springer Verlag, 2001.