WCET Code Positioning

Wankang Zhao[1], David Whalley[1], Christopher Healy[2], Frank Mueller[3]
[1]Computer Science Dept., Florida State University, Tallahassee, FL 32306-4530; e-mail: whalley@cs.fsu.edu
[2]Computer Science Dept., Furman University, Greenville, SC 29613; e-mail: chris.healy@furman.edu
[3]Computer Science Dept., North Carolina State University, Raleigh, NC 27695; e-mail: mueller@cs.ncsu.edu

Abstract

Some processors incur a pipeline delay whenever an instruction transfers control to a target that is not the next sequential instruction. Compiler writers attempt to reduce these delays by positioning the basic blocks within a function to minimize the number of unconditional jumps and taken conditional branches that occur. Such a code positioning algorithm is traditionally driven by profile data representing typical program executions where pairs of blocks are placed in contiguous order when the transitions between these blocks occur most frequently. In this paper we describe an approach to perform code positioning without profiling in an attempt to reduce WCET instead of ACET. Our compiler interacts with a timing analyzer to obtain WCET path information to guide the block positioning. The results show over a 9% average reduction in WCET is achieved after code positioning is performed and our greedy WCET code positioning algorithm always achieves optimal results for our benchmark suite.

1. Introduction

Generating acceptable code for applications residing on embedded systems is challenging. Unlike most general-purpose applications, embedded applications often have to meet various stringent constraints, such as time, space, and power. Constraints on time are commonly formulated as *worst-case* (WC) constraints. If these timing constraints are not met, even only occasionally in a hard real-time system, then the system may not be considered functional.

The *worst-case execution time* (WCET) must be calculated to determine if a timing constraint will always be met. Accurate and safe WCET predictions can only be obtained by a tool that statically analyzes an application to calculate an estimated WCET. Such a tool is called a *timing analyzer*, and the process of performing this calculation is called *timing analysis*.

It is desirable to not only accurately predict the WCET, but to also improve it. Improving the WCET of a task may enable an embedded system to meet its timing constraints that were previously infeasible. Improving the WCET may also allow an embedded system developer to use a lower clock rate (still meeting the timing constraints) and save power, which is valuable for mobile applications.

One type of compiler optimization is to reorder or position the basic blocks within a function. The benefits of such a transformation include improving instruction cache locality and reducing misfetch penalties. In recent years instruction cache performance has become less of a concern as instruction caches have increased in size. In fact, many embedded processors have no instruction cache and an embedded application is instead often placed in ROM. However, some processors still incur a pipeline delay associated with each transfer of control. Such delays are more common for embedded machines where branch prediction and target buffers may not exist in order to reduce the complexity of the processor. Compiler writers attempt to reduce these delays by ordering the basic blocks to minimize the number of unconditional jumps and taken branches that occur. The optimization phase that performs this transformation in a compiler is typically referred to as a code positioning or branch alignment optimization. Existing code positioning algorithms weight the directed edges (transitions) between the nodes (basic blocks) of a control-flow graph by the number of times the edge was traversed at run-time [1, 2, 3, 4, 5]. In general, these algorithms order basic blocks by attempting to make the most frequently traversed edges contiguous in memory.

Unfortunately, traditional code positioning algorithms are not guaranteed to reduce the WCET of an application since the most frequently executed edges may not be contained in the WC paths. Even if WCET path information were used by the code positioning algorithm, a change in the positioning may result in a different path becoming the WC path in a loop or a function. In contrast, the frequency of the edges based on profile data, which is used in traditional code positioning, does not change regardless of how the basic blocks are ordered. Thus, WCET code positioning is inherently more challenging than ACET (average case execution time) code positioning.

In this paper we describe an approach for improving the WCET of an application by code positioning. We have integrated a timing analyzer with a compiler where the WCET of the application and the current function can be calculated on demand. After each edge is selected for positioning, the timing analyzer is invoked and up-to-date WCET path information is obtained. After positioning all of the edges in a function, we also align the blocks which are targets of transfers of control to further reduce WCET by minimizing target misalignment penalties.

2. Related Work

A timing analyzer is a tool that can statically analyze a program and predict its WCET. There have been a number of general approaches that have been used to develop timing analyzers. One approach to predict WCET uses ILP (integer linear programming). An *ILP* based approach examines the control flow of a program and derives constraints that can be input to an ILP solver to predict the WCET [6, 7, 8]. This approach is appealing since constraints due to the structure of the control flow graph (structural) and program values (functional) can both be provided to the same solver. However, the result of the ILP analysis is a single WCET prediction for the entire task, which means that it does not provide the detailed information needed by a compiler to optimize for WCET. The ILP approach can also become more time consuming as the size of the program and the corresponding number of constraints increase. Other researchers have used a *symbolic-execution* approach to calculate the WCET [9, 10]. This approach symbolically simulates the program instructions by allowing the values of variables to be unknown. Symbolic execution can provide a very accurate WCET estimate since it can implicitly handle most functional constraints. However, the analysis time is proportional to the WC number of executed instructions. Thus, the analysis can be prohibitively slow, which is not ideal for interfacing with a compiler. In contrast, we use a *path-based* approach for timing analysis, where each path at each function and loop level is analyzed for its WCET [11, 12, 13, 14, 15, 16, 17, 18, 19]. Our timing analyzer also calculates WCET predictions very quickly [19], which means performing timing analysis during compilation would not significantly increase compilation time. Thus, it is feasible to use a path-based timing analysis approach to supply WCET path information to a compiler in an attempt to reduce the WCET of a function.

While there has been much work on developing compiler optimizations to reduce execution time and, to a lesser extent, to reduce space and power consumption, there has been very little work to reduce WCET. Marlowe and Masticola outlined how a variety of standard compiler optimizations could potentially affect timing constraints of critical portions in a task. However, no implementation was described [20]. Hong and Gerber developed a programming language with timing constructs and used a trace scheduling approach to improve code in what they deemed to be critical sections of the program. However, no empirical results were given since the implementation did not interface with a timing analyzer to evaluate the impact on reducing WCET [21]. Both of these papers outlined strategies to move code outside of critical sections within an application that have been designated to contain timing constraints. However, most real-time systems use the WCET of entire tasks to determine if a schedule can be met. Lee *et al.* used WCET information to select how to generate code on a dual instruction set processor for the ARM and the Thumb [22]. ARM code is generated for a selected subset of basic blocks that can impact the WCET. Thumb code is generated for the remaining blocks to minimize code size. In contrast, we have developed a compiler optimization to reduce the WCET of an application on a single instruction set processor. Finally, a genetic algorithm has been used to search for an effective optimization phase sequence that best reduces WCET for an application [23]. This approach uses standard compiler optimizations, whereas we have developed optimizations that are driven by WCET path information.

There have been several code positioning (basic block reordering) approaches that have been developed. Some algorithms have the goal of improving instruction cache performance [1, 2, 3]. Other algorithms have the primary goal of reducing the number of dynamic transfers of control (e.g. unconditional jumps and taken branches) and the associated pipeline penalty on specific processors [4, 5]. All of these approaches use profile information to obtain a weight for each directed edge in a control-flow graph by counting the number of times the edge was traversed at run-time. Thus, these approaches attempt to improve ACET. In contrast, we describe a code positioning algorithm in this paper to improve WCET. Other approaches have performed code positioning in combination with code duplication to avoid the execution of unconditional jumps and branches [24, 25]. The benefit of these approaches comes at the expense of increased code size, which may not be appropriate for embedded applications.

3. Experimental Environment

In this section we provide a brief description of the experimental environment in which this research was performed. We give an overview of the compiler and timing analyzer that we use and how they interact. We also

describe the processor for which the compiler generates code and the timing analyzer calculates the WCET.

We have developed a system where a compiler can obtain WCET information from a timing analyzer upon demand [23]. Figure 1 shows an overview of the flow of information. The compiler will send information about the control flow and the current instructions that have been generated to the timing analyzer. WCET predictions will be sent back to the compiler. The compiler we use is VPO, which performs its optimizations on a low level representation that is equivalent to machine instructions [26]. This level is appropriate for interfacing with a timing analyzer so that accurate WCETs can be obtained.

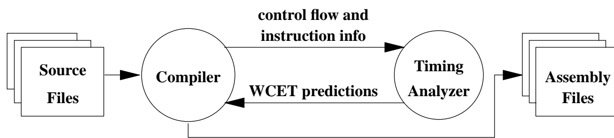

Figure 1: Overview of the Worst-Case Aware Compilation Process

The timing analyzer we use for this study calculates the WCET for each path, loop, and function in the program. It performs this analysis in a bottom up fashion, where the WCET for an inner loop (or called function) is calculated before determining the WCET for an outer loop (or calling function). Our timing analyzer has been used in the past to predict WCETs for applications that execute on machines with an instruction cache [11, 15], a pipeline [12, 15], and a data cache [13, 17]. In addition, it can automatically calculate the maximum number of iterations of many loops, including those involving nonrectangular loop nests [14, 18]. Finally, the timing analyzer can also detect many constraints on branches that restrict the set of paths that can be taken in a program [16, 19].

We have ported both the VPO compiler and our timing analyzer to the StarCore SC100 processor [27]. This processor has neither a memory hierarchy (no caches or virtual memory system) nor an OS, which facilitates obtaining tight WCET predictions [23]. It has no architectural support for floating-point operations since it is a digital signal processor and was designed instead for fixed-point arithmetic. It has 16 data registers and 16 address registers. The SC100 also has a simple five stage pipeline, where most instructions can perform its execution in a single stage. There are no pipeline interlocks and it is the responsibility of the compiler to schedule instructions and to insert noops when a subsequent instruction uses the result of a preceding instruction that will not be available in the pipeline. The size of the instructions can vary from one word (two bytes) to five words (ten bytes) depending on the instruction type, addressing modes used, and register numbers that are referenced.

The most relevant feature of the SC100 for WCET code positioning is that all transfers of control (taken branches, unconditional jumps, calls, and returns) result in a one to three cycle penalty depending on the addressing mode used and if a transfer of control uses a delay slot. We have found that transfers of control penalties can lead to nonintuitive WCET results. Consider the flow graph in Figure 2. A superficial inspection of the corresponding function might lead one to believe that the path 1→2→3 is the WCET path since it results in more instructions being executed than in the path 1→3. However, if the taken branch penalty in path 1→3 outweighs the cost of executing the instructions in block 2, then path 1→3 would be the WCET path. This simple example illustrates the importance of using a timing analyzer to calculate the WCET. Measuring the execution time is not safe since it is very difficult to manually determine both the WC paths and the input data to drive the execution of these paths.

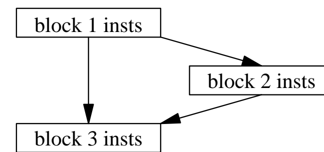

Figure 2: Example Control-Flow Graph

Another particularly relevant feature of the SC100 for WCET code positioning is that SC100 instructions are grouped into fetch sets, which are four words (eight bytes) in size and are aligned on eight byte boundaries. When a transfer of control occurs to an instruction in a new fetch set and the target instruction spans more than one fetch set, then the processor stalls for an additional cycle. This situation is illustrated in Figure 3, where the target instruction spans fetch sets *n+1* and *n+2*.

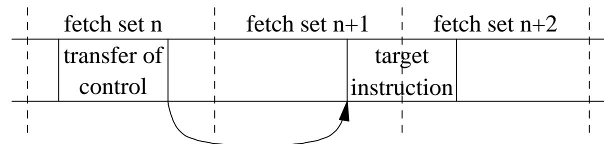

Figure 3: Example of a Misaligned Target Instruction

4. WCET Code Positioning

The goal of most code positioning algorithms is to reduce the ACET by positioning the basic blocks within the frequent flow of control contiguously in memory. Code positioning is essentially an attempt to find the most efficient permutation of the basic blocks in a function. Exhaustive approaches are not typically feasible except

when the number of blocks is small since there are $n!$ possible permutations, where n is the number of basic blocks in the function. Thus, most approaches use a greedy algorithm to avoid excessive increases in compilation time.

The goal of a WCET positioning algorithm is to select edges between blocks to be contiguous that will minimize the WCET. A directed edge connecting two basic blocks is *contiguous* if the source block is immediately followed by the target block in memory. However, not all edges can be contiguous. Consider the portion of a control-flow graph shown in Figure 4. If edge *b* (shown as a solid line) is selected to be contiguous, then no other edges to the same target can be contiguous. For example, edge *a* can no longer be contiguous since its source block 4 cannot be positioned immediately before its target block 2. Likewise, only a single edge among the set that share the same source block can contiguous. For instance, selecting edge *b* to be contiguous will make edge *c* noncontiguous since the target block 3 cannot be positioned immediately after source block 1.

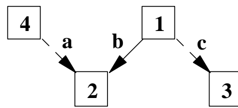

Figure 4: Selecting an Edge to Be Contiguous

WCET code positioning needs to be driven by WCET path information. Our timing analyzer calculates all paths within each loop and the outer level of a function. A path consists of nodes that are basic blocks and edges that are control-flow transitions. Each loop path starts with the entry block in the loop and is terminated by a block that has a transition back to the entry block or outside the loop. A function path starts with the entry block to the function and is terminated by a block containing a return. If a path enters a nested loop, then the entire nested loop is considered a single node along that path.

Our compiler obtains the WCET for each path in the function from the timing analyzer. If the timing analyzer calculates the WCET path information on the original positioned code, then changing the order of the basic blocks may result in unanticipated increases in the WCET for other paths since previously contiguous edges may become noncontiguous. We decided instead to treat the basic blocks as being initially unpositioned. Thus, we actually modify the code so that all transitions between blocks are accomplished using a transfer of control and will result in a transfer of control penalty. This means an unconditional jump is added after each basic block that does not already end with an unconditional transfer of control (i.e., unconditional jump or return).

Consider the source code in Figure 5, which is a contrived example to illustrate the algorithm. Figure 6 shows the corresponding control flow that is generated by the compiler. While the control flow in the figure is represented at the source code level to simplify its presentation, the analysis is performed by the compiler at the assembly instruction level after compiler optimizations are applied to allow more accurate timing predictions. Note that some branches in Figure 6 have conditions that are reversed from the source code in Figure 5 to depict the branch conditions that are evaluated at the assembly instruction level. Several unconditional jumps, represented in Figure 6 as ***goto*** statements underneath dashed lines, have been inserted to make all transitions between basic blocks result in a transfer of control penalty. The unconditional jumps in blocks 3 and 6 were already present. Conditional branches are represented as ***if*** statements in Figure 6. The jumps (shown as ***goto*** statements) immediately following each conditional branch are actually placed in separate basic blocks within the compiler's representation, but are shown in the same block as the corresponding branch in the figure to simplify the presentation of the example. The transitions (directed edges) between nodes are labeled so they can be referenced later. Figure 7 shows the paths through the control flow graph. Paths A-D represent paths within the loop. Path E represents the outer level path, where the loop is considered a single node within that path. We consider backedges (directed edges back to the entry point of the loop) to be part of the paths within the loop since these edges can be traversed on all loop iterations, except for the last one. Likewise, the exit edges (directed edges leaving the loop) are considered part of the outer paths containing the loop since an exit edge is executed at most once each time the loop is entered.

```
...
for (i = 0; i < 1000; i++) {
   if (a[i] < 0)
      a[i] = 0;
   else {
      a[i] += 1;
      sumalla += a[i];
   }
   if (b[i] < 0)
      b[i] = 0;
   else {
      b[i] += 1;
      sumallb += b[i];
      b[i] = a[i] - 1;
   }
}
...
return;
```

Figure 5: Source Code Example

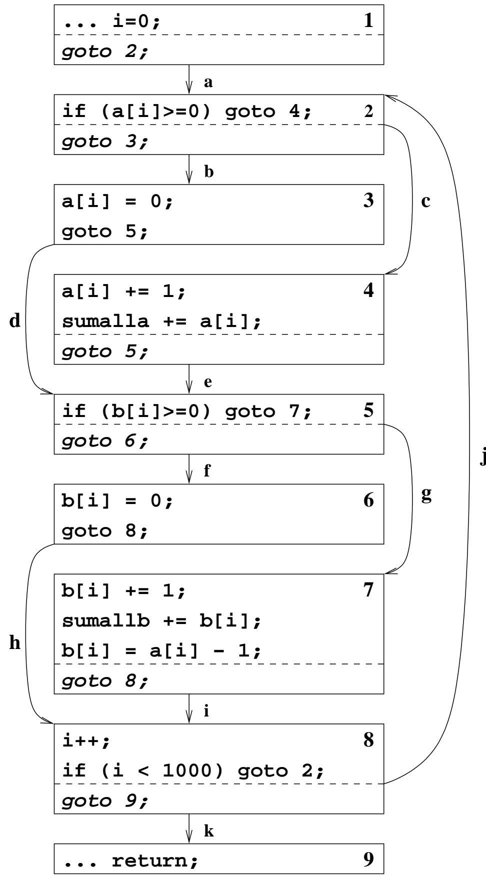

Figure 6: Control Flow Graph of Code in Figure 5

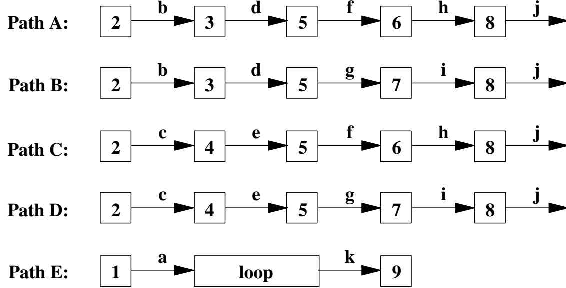

Figure 7: Paths in Figure 6

There are a few terms that need to be defined before our WCET code positioning algorithm can be presented. Edges are denoted as being contiguous, noncontiguous, or unpositioned. A *contiguous* edge has its source block immediately positioned before its target block in memory. In contrast, a *noncontiguous* edge does not. An *unpositioned* edge means that it has not yet been determined if it will be contiguous or noncontiguous. The UB-WCET (upper bound WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be noncontiguous. The LB-WCET (lower bound WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be contiguous. Paths are also classified as *contributing* or *noncontributing* to the WCET. A path is considered *noncontributing* when its UB-WCET is less than the LB-WCET of another path within the same loop (or outermost level of a function). Noncontributing paths cannot affect the WCET.

Our WCET code positioning algorithm is described in Figure 8. At this point target alignment penalties are not assessed by the timing analyzer since WCET target alignment, described in Section 5, is performed after WCET code positioning. The algorithm selects one *unpositioned* edge at a time to make *contiguous*. These edges are selected by first examining the paths that most affect the WCET. Thus, we also weight paths by the maximum number of times that they can be executed in the function to ensure its effect on the WCET is accurately represented.[1] After selecting an edge to be *contiguous* (and possibly making one or more other edges *noncontiguous*), the UB-WCET and LB-WCET of each path are recalculated. The algorithm continues until all edges have been positioned.

Table 1 shows how WCET code positioning is accomplished for the example shown in Figures 5, 6, and 7. At each step the status for each edge and the current UB-WCET and LB-WCET for each path calculated from the timing analyzer are shown. Initially all edges are unpositioned, as shown in step 0. For each step an edge is selected to be *contiguous* and one or more edges become *noncontiguous*. Thus, after each step one or more paths have their UB-WCET reduced and one or more paths have their LB-WCET increased. In the first step, the algorithm selects edge *j* to be *contiguous* since it reduces the UB-WCET of all four paths in the loop. This selection also causes edges *a* and *k* to become *noncontiguous*, which results in only a small increase for the LB-WCET of the entire function (path *E*) since these edges are outside the loop. In the second step, edge *i* is selected since it is part of path *D*, which contains the greatest current UB-WCET. The algorithm chooses edge *i* instead of another edge in path *D* since edge *i* is also part of path *B*, which contains the second greatest WCET at that point. Edge *g* is selected to be *contiguous* in the third step since that is also part of path *D*, which still contains the greatest UB-WCET. Edge *e* becomes *contiguous* in the fourth step

[1] Different loops may have a different maximum number of iterations. Our timing analyzer automatically determines the number of iterations for each loop in the function [14, 18]. In addition, the number of iterations in which a path may be executed can be restricted due to constraints on branches [16, 19].

(1) Add an unconditional jump at the end of each basic block that does not end with an unconditional transfer of control. Mark all edges as *unpositioned*.

(2) Invoke the timing analyzer to calculate the UB-WCET and LB-WCET path information. The LB-WCET path information is calculated by not including jumps or taken branch penalties associated with *unpositioned* edges.

(3) Sort the paths in the function in descending order based on first if it is *contributing*, next the number of times it can be executed that is the product of the number of iterations of each loop in which it is nested, next its UB-WCET, and finally its LB-WCET. Thus, edges in *contributing* paths will be addressed first since *noncontributing* edges cannot affect the WCET.

(4) If all of the edges in the function have been positioned, then go to step 8.

(5) Select the first path in the list that has at least one *unpositioned* edge.

(6) Choose the edge within this path to make *contiguous* that minimizes the UB-WCET of this path. Break ties by selecting the edge that produces a lower UB-WCET of a path that is first encountered in the sorted list. Break remaining ties by selecting the edge that results in a smaller increase in the LB-WCET of a path that is first encountered in the sorted list.

(7) Mark the chosen edge as *contiguous*. All paths that include this edge will have their UB-WCET reduced. Mark other edges that become *noncontiguous* as a result of choosing the *contiguous* edge. All paths that contain these edges will have their LB-WCET increased. Go to step 2.

(8) Connect the *contiguous* edges with common nodes to perform the final positioning of the basic blocks.

Figure 8: WCET Code Positioning Algorithm

since it is part of path *C*, which currently contains the greatest UB-WCET. At this point path *D*'s UB-WCET becomes 29, which is less than the LB-WCET of 30 for path *A*. Thus, path *D* is now *noncontributing*. During the fifth step edge *b* is selected since it is part of path *A*, which contains the current greatest UB-WCET. At this point all of the edges have been positioned and the UB-WCET and LB-WCET for each path are now identical. The original positioning shown in Figure 6, but without the extra jumps inserted to make all transitions noncontiguous, has a WCET of 31,018 or about 14.8% greater than after WCET code positioning.

While the edges have been positioned according to the selections shown in Table 1, the final positioning of the basic blocks still has to performed. The list of contiguous edges in the order in which they were selected are 8→2, 7→8, 5→7, 4→5, and 2→3. Connecting these edges by their common nodes, we are able to determine that six of the nine blocks should be positioned in the order 4→5→7→8→2→3. The remaining blocks, which are 1, 6, and 9, can be placed either before or after this contiguous set of blocks. In general, there may be several contiguous sets of blocks in a function and these sets can be placed in an arbitrary order. We always designate the

Table 1: Information for Each Step of the Algorithm in Figure 8 for the Example Shown in Figures 5, 6, and 7

Step	Status of Edges Shown in Figure 6											WCETs of Paths Shown in Figure 7									
												UB-WCET					LB-WCET				
	a	b	c	d	e	f	g	h	i	j	k	A	B	C	D	E	A	B	C	D	E
0	u	u	u	u	u	u	u	u	u	u	u	36	40	37	41	37,020	21	25	22	26	22,018
1	n	u	u	u	u	u	u	u	u	c	n	33	37	34	38	34,024	21	25	22	26	22,024
2	n	u	u	u	u	u	u	n	c	c	n	33	34	34	35	31,024	24	25	25	26	22,024
3	n	u	u	u	u	n	c	n	c	c	n	33	31	34	32	30,024	27	25	28	26	24,024
4	n	u	u	n	c	n	c	n	c	c	n	33	31	31	29	29,024	30	28	28	26	26,024
5	n	c	n	n	c	n	c	n	c	c	n	30	28	31	29	27,024	30	28	31	29	27,024

u = unpositioned, c = contiguous, n = noncontiguous

entry block of the function as the first block in the final positioning to simplify the generation of the assembly code by our compiler and the processing by our timing analyzer. Note that the entry block can never be the target of an edge in the control flow due to prologue code for the function being generated in this block. By contrasting the code in Figure 6 with the final positioning in Figure 9, one can observe that performing the final positioning sometimes requires reversing branch conditions, changing target labels of branches, labeling blocks that are now targets of branches or jumps, inserting new unconditional jumps, and deleting other jumps. All of the loop paths, *A*-*D*, prior to WCET code positioning required three transfers of control. After WCET code positioning paths A and C each require three transfers of control and paths *B* and *D* each require only one. Note that paths *B* and *D* had higher UB-WCETs before the edges were positioned.

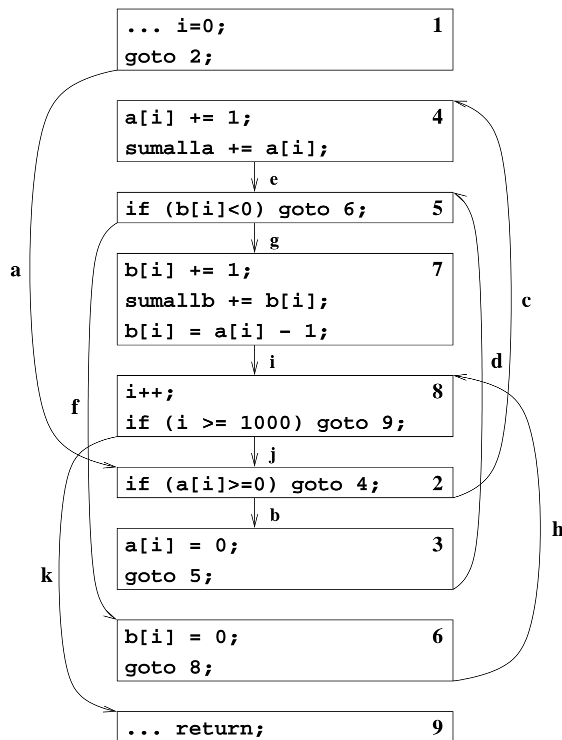

Figure 9: Control Flow Graph of Code in Figure 5 after WCET Positioning

The portion of the greedy algorithm shown in Figure 8 that most affects the analysis time is the computation performed by the timing analyzer, which is invoked each time an edge is selected to become contiguous. Given that there are n basic blocks in a function, there can be at most n-1 contiguous edges and sometimes there are less. For instance, only five edges were selected to be contiguous instead of n-1 or eight edges for the example shown in Table 1 and Figure 9. Thus, the timing analyzer is invoked at most n-1 times for each function, which is much less than the $n!$ invocations that would be required if every possible basic block ordering permutation was checked.

5. WCET Target Alignment

Even after the basic blocks have been positioned within a function, there can still be extra transfer of control penalties due to misaligned targets. The SC100 fetches instructions in sets of four words that are aligned on eight byte boundaries. The target of a transfer of control is considered misaligned when the target instruction is in a different fetch set from the transfer of control and the target instruction spans more than one fetch set, as shown previously in Figure 3. In this situation, the processor stalls for an additional cycle after the transfer of control.

We attempt to minimize the number of target misalignment penalties in the following manner. First, we partition the function into relocatable sets of basic blocks. The first block in a relocatable set is not fallen into from a predecessor block and the last block ends with an unconditional transfer of control, such as an unconditional jump or a return. A relocatable set of blocks can be moved without requiring the insertion of additional instructions. For instance, the code in Figure 9 after WCET positioning has four relocatable sets of blocks, which are {1}, {4,5,7,8,2,3}, {6}, and {9}. In contrast, the original flow graph of blocks in Figure 6 without the additional unconditional jumps to make edges *unpositioned* has three relocatable sets, which are {1,2,3}, {4,5,6}, and {7,8,9}. After WCET code positioning we align the relocatable sets of blocks one set at a time. Since each instruction has to be aligned on a word boundary (2 bytes) and each fetch set consists of 4 words, we try four different positionings for each set. The different alignments are accomplished by inserting 0, 1, 2, or 3 noops before the beginning of the relocatable set, where each noop is one word in size. The insertion of noops before each relocatable set of blocks is illustrated in Figure 10. Note that these noop instructions are not reachable in the control flow and are never executed. We invoke the timing analyzer to determine the WCET of the function with each combination. Thus, we have to invoke the timing analyzer m*4 times, where m is the number of relocatable sets of blocks to be aligned. We choose the number of noops to insert before each set that will minimize the WCET for the function. To help support this analysis, we added an option to the timing analyzer to only assess misalignment penalties within a range of blocks where target alignment has been performed. In the case that the WCET is the same for two or more options, we select the option with the fewest noops.

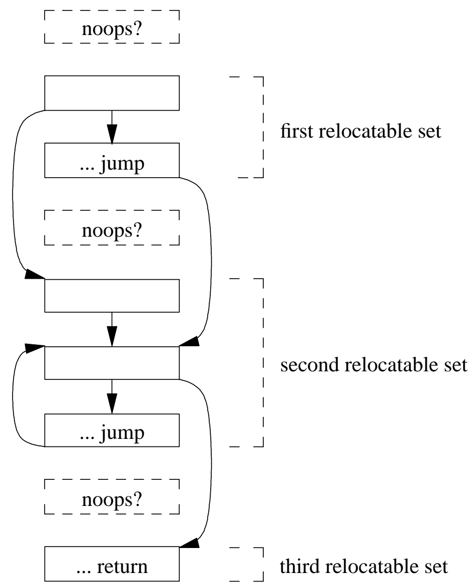

Figure 10: Example of Inserting Noop Instructions before Each Relocatable Set of Blocks

We could attempt a more aggressive approach by trying all permutations of ordering relocatable sets of blocks in addition to inserting noops. This approach could potentially reduce the number of noops inserted. However, we have found that the code size increase is small and our current approach is quite efficient.

6. Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of improving the WCET by performing WCET code positioning and WCET target alignment. Table 2 shows the benchmarks we used for our experiments. The benchmark *sumposclrneg* contains the example code shown in Figure 5. The other benchmarks were selected since they have conditional constructs and most have been used in previous studies by various groups (FSU, SNU, Uppsala) working on WCET timing analysis.

Table 3 shows the accuracy of our timing analyzer and the effect on WCET after code positioning and target alignment. The results *before WCET positioning* indicate the measurements taken after all optimizations have been applied except for WCET code positioning and WCET target alignment. The *observed cycles* were obtained from running the compiled programs through the SC100 simulator [28] using WCET input data.[2] All input and output were accomplished by reading from and writing to global variables, respectively, to avoid having to estimate the WCET of performing actual I/O. The *WCET cycles* are the WCET predictions obtained from our timing analyzer. The *WCET ratio* is the *WCET cycles* divided by the *observed cycles*. In general, our timing analyzer is able to obtain tight WCET predictions for code generated for the SC100. The results *after WCET positioning* indicate the measurements taken after the positioning algorithm described in Section 4 is applied immediately following the preceding optimization phases. The *WCET cycles* represent the new predicted WCET by the timing analyzer. The *positioning ratio* indicates the ratio of the *WCET cycles after WCET positioning* divided by the *WCET cycles before WCET positioning*. There was over a 9% average reduction in WCET by applying the WCET code positioning algorithm. The results *after WCET alignment* indicate the measurements that were obtained after the WCET target alignment algorithm in Section 5 is applied following WCET code positioning. The *WCET cycles* again represent the new predicted WCET by the timing

[2] The WCET input data had to be meticulously determined since the WCET paths were often difficult to detect manually due to control-flow penalties, as illustrated in Figure 2. We did not obtain the *observed cycles* after WCET positioning or WCET alignment since this would require new WCET input data due to changes in the WCET paths.

Table 2: Benchmarks Used in the Experiments

Program	Description
bubblesort	performs a bubble sort on 500 elements
keysearch	performs a linear search involving 4 nested loops for 625 elements
summidall	sums the middle half and all elements of a 1000 integer vector
summinmax	sums the minimum and maximum of the corresponding elements of two 1000 integer vectors
sumoddeven	sums the odd and even elements of a 1000 integer vector
sumnegpos	sums the negative, positive, and all elements of a 1000 integer vector
sumposclrneg	sums positive values from two 1000 element arrays and sets negative values to zero
sym	tests if a 100x100 matrix is symmetric
unweight	converts an adjacency 100x100 matrix of a weighted graph to an unweighted graph

Table 3: Results after WCET Code Positioning and Target Alignment

Program	Before WCET Positioning			After WCET Positioning		After WCET Alignment		Time Ratio
	Observed Cycles	WCET Cycles	WCET Ratio	WCET Cycles	Positioning Ratio	WCET Cycles	Alignment Ratio	
bubblesort	7,497,532	7,748,545	1.033	7,747,045	**1.000**	7,622,296	**0.984**	1.94
keysearch	30,667	31,143	1.016	29,268	**0.940**	29,268	**0.940**	2.17
summidall	19,508	19,515	1.000	16,721	**0.857**	16,721	**0.857**	1.33
summinmax	24,010	24,015	1.000	22,021	**0.917**	21,023	**0.875**	1.56
sumnegpos	20,010	20,015	1.000	18,021	**0.900**	18,021	**0.900**	1.33
sumoddeven	22,021	23,027	1.046	18,030	**0.783**	16,543	**0.718**	1.67
sumposclrneg	31,013	31,018	1.000	27,024	**0.871**	27,024	**0.871**	1.90
sym	223,168	223,472	1.001	208,622	**0.934**	208,622	**0.934**	1.70
unweight	350,507	350,814	1.001	341,020	**0.972**	341,020	**0.972**	2.00
average	913,160	941,285	1.011	936,419	**0.908**	922,282	**0.895**	1.73

analyzer. The *alignment ratio* indicates the ratio of the *WCET cycles after WCET alignment* as compared to the *WCET cycles before WCET positioning*. Three of the nine benchmarks improved due to WCET target alignment, which resulted in over an additional 1% average improvement in WCET. The *time ratio* indicates the compilation overhead of performing WCET code positioning and target alignment. Most of this overhead is due to repeated calls to the timing analyzer. While this overhead is reasonable, it could be significantly reduced if the timing analyzer and the compiler were in the same executable and passed information via arguments instead of files.

While the results in Table 3 show a significant improvement in the predicted WCET, it would be informative to know if better positionings than those obtained by our greedy WCET code positioning algorithm are possible. Like most benchmarks used for WCET prediction studies, the size of each benchmark is fairly small so that the WCET input data can be manually determined and the WCET observed cycles can be measured. The functions in these benchmarks were small enough so that the WCET for every possible permutation of the basic block ordering could be estimated. The number of possible orderings for each function is $n!$, where n is the number of basic blocks, since each block can be represented at most once in the ordering. Table 4 shows the results of performing an exhaustive search for the best WCET code positioning for each benchmark, where the WCET is calculated for each possible permutation. Unlike the measurements shown in Table 3, these WCET results exclude target misprediction penalties. Our WCET positioning algorithm does not take target misprediction penalties into account when making positioning decisions since the WCET target alignment optimization occurs after positioning. Thus, the WCETs are in general slightly lower than the WCETs shown in Table 3. The number of *permutations* varied depending upon the number of *routines* in the benchmark and the number of basic blocks in each function. The *minimum WCET* represents the lowest WCET found by performing the exhaustive search. There are typically multiple code positionings that result in an equal *minimum WCET*. We found that the *greedy WCET* obtained by our algorithm was always identical to the *minimum WCET* for each function in each benchmark for our test suite. While we cannot always guarantee an optimal result, it does appear that our greedy algorithm is very effective at finding an efficient WCET code positioning. The *default WCET* represents the WCET with the default code positioning. On average the *default WCET* is 11.6% worse than the *minimum WCET*. The *maximum WCET* represents the highest WCET found during the exhaustive search. The results show that the *maximum WCET* is 50.2% higher on average than the *minimum WCET*. While the *default WCET* is relatively efficient compared to the *maximum WCET*, the *greedy WCET* still is a significant improvement over just using the default code positioning.[3]

[3] Invoking the timing analyzer $n!$ times when performing an exhaustive search for each function would require an excessive amount of time. Instead, we initially invoked the timing analyzer once without assessing transfer of control penalties to obtain a base WCET time for each path. For each permutation we adjusted each path's WCET by adding the appropriate transfer of control penalty to each noncontiguous edge. After finding the minimum WCET permutation, we invoked the timing analyzer again for this permutation to verify that our preliminary WCET prediction without using the timing analyzer was accurate. While this approach is potentially less accurate, we were able to obtain results in a few hours. Invoking the timing analyzer for each permutation would have taken significantly longer.

Table 4: Minimum, Greedy, Default, and Maximum WCET Code Positioning Results

Program	Routines	Permutations	Minimum WCET	Greedy WCET	Greedy Ratio	Default WCET	Default Ratio	Maximum WCET	Maximum Ratio
bubblesort	4	40,328	7,747,045	7,747,045	**1.000**	7,748,545	1.000	9,248,542	1.194
keysearch	2	39,916,801	29,238	29,238	**1.000**	31,113	1.064	59,574	2.038
summidall	1	5,040	16,721	16,721	**1.000**	18,515	1.107	28,721	1.718
summinmax	1	362,880	21,021	21,021	**1.000**	24,015	1.142	30,021	1.428
sumnegpos	1	5,040	18,021	18,021	**1.000**	20,015	1.111	24,021	1.333
sumoddeven	1	3,628,800	16,029	16,029	**1.000**	22,044	1.375	31,044	1.937
sumposclrneg	1	362,880	27,024	27,024	**1.000**	31,018	1.148	37,024	1.370
sym	2	5041	208,622	208,622	**1.000**	223,472	1.071	238,922	1.145
unweight	1	40,320	341,020	341,020	**1.000**	350,714	1.028	461,320	1.353
average	1.5	5,540,851	936,082	936,082	**1.000**	941,050	1.116	1,128,799	1.502

7. Future Work

The WCET code positioning algorithm could be adapted to improve WCETs on processors with an instruction cache. Improving instruction cache performance would require the optimization to be performed at a later stage in the compilation process where all of the application code is accessible. We currently perform WCET code positioning during the compilation of a single function.

So far we have always obtained an optimal positioning using our WCET code positioning algorithm. It may be that the algorithm for a processor such as the SC100 is actually optimal and it would be interesting to attempt to prove the optimality of the algorithm.

There are also a number of other compiler optimizations that can be redesigned to improve WCET as opposed to improving ACET. Rather than improving the frequently executed portions of the program, we can use WCET information to determine the portions that contribute the most to the WCET. For instance, there have also been many compiler optimizations that have been developed to improve the code within frequently executed paths. These optimizations can be redesigned to instead improve the WC path. Similar to issues faced in WCET code positioning, the compiler will have to consider that the WCET path may change as optimizations are applied.

8. Conclusions

In this paper we have described a code positioning algorithm that is driven by WCET path information, as opposed to ACET frequency data. Our algorithm addresses the issue of causing a different path to increase its WCET and become the WC path after changing the positioning. We accomplish this by initially assuming that all edges are unpositioned. At each step we conservatively estimate the WC paths in the function based on the currently unpositioned edges and use this information to select the next edge to make contiguous. We also determine which paths cannot contribute to the WCET by calculating when its UB-WCET is less than the LB-WCET of another path in the same loop or outermost level of a function. Edges that appear in only noncontributing paths have the lowest priority for being made contiguous. We have implemented the algorithm and have demonstrated that it can improve the WCET of applications on a machine which has transfer of control penalties. In fact, we showed that our greedy WCET code positioning algorithm obtains optimal results on the SC100 for our test suite of programs. We have also implemented and evaluated a related compiler optimization to reduce WCET due to target misalignment penalties. Thus, we have shown it is feasible to develop specific compiler optimizations that are designed to improve WCET using WCET path information as opposed to improving ACET using frequency data.

9. Acknowledgements

The anonymous reviewers' suggestions improved the quality of the paper. We also thank StarCore for providing the necessary software and documentation that were used in this project. This research was supported in part by NSF grants EIA-0072043, CCR-0208581, CCR-0208892, CCR-0310860, CCR-0312493, and CCR-0312695.

10. References

[1] S. McFarling, "Program Optimization for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191 (April 1989).

[2] W. W. Hwu and P. P. Chang, “Achieving High Instruction Cache Performance with an Optimizing Compiler,” *Proceedings of the 16th Annual Symposium on Computer Architecture*, pp. 242-250 (May 1989).

[3] K. Pettis and R. Hansen, “Profile Guided Code Positioning,” *Proceedings of the SIGPLAN ’90 Conference on Programming Language Design and Implementation*, pp. 16-27 (June 1990).

[4] B. Calder and D. Grunwald, “Reducing Branch Costs via Branch Alignment,” *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 242-251 (October 1994).

[5] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith, “Near-optimal Intraprocedural Branch Alignment,” *Proceedings of the SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pp. 183-193 (June 1997).

[6] Y. S. Li, S. Malik, and A. Wolfe, “Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 298-307 (December 1995).

[7] H. Theiling, C. Ferdinand, and R. Wilhelm, “Fast and Precise WCET Prediction by Separate Cache and Path Analyses,” *Real-Time Systems* **18**(May 2000).

[8] J. Engblom and A. Ermedahl, “Modeling Complex Flows for Worst-Case Execution Time Analysis,” *Proceedings of the IEEE Real-Time Systems Symposium*, (December 2000).

[9] T. Lundqvist and P. Stenström, “Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1-15 (June 1998).

[10] T. Lundqvist and P. Stenström, “An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution,” *Real-Time Systems* **17** pp. 183-207 (November 1999).

[11] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, “Bounding Worst-Case Instruction Cache Performance,” *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 172-181 (December 1994).

[12] C. A. Healy, D. B. Whalley, and M. G. Harmon, “Integrating the Timing Analysis of Pipelining and Instruction Caching,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 288-297 (December 1995).

[13] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, “Timing Analysis for Data Caches and Set-Associative Caches,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 192-202 (June 1997).

[14] C. A. Healy, M. Sjödin, V. Rustagi, and D. B. Whalley, “Bounding Loop Iterations for Timing Analysis,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 12-21 (June 1998).

[15] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon, “Bounding Pipeline and Instruction Cache Performance,” *IEEE Transactions on Computers* **48**(1) pp. 53-70 (January 1999).

[16] C. A. Healy and D. B. Whalley, “Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 79-88 (June 1999).

[17] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, “Timing Analysis for Data Caches and Wrap-Around Fill Caches,” *Real-Time Systems*, pp. 209-233 (November 1999).

[18] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen, “Supporting Timing Analysis by Automatic Bounding of Loop Iterations,” *Real-Time Systems*, pp. 121-148 (May 2000).

[19] C. Healy and D. Whalley, “Automatic Detection and Exploitation of Branch Constraints for Timing Analysis,” *IEEE Transactions on Software Engineering* **28**(8) pp. 763-781 (August 2002).

[20] T. Marlowe and S. Masticola, “Safe Optimization for Hard Real-Time Programming,” *System Integration*, pp. 438-446 (June 1992).

[21] S. Hong and R. Gerber, “Compiling Real-Time Programs into Schedulable Code,” *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, pp. 166-176 (June 1993).

[22] S. Lee, J. Lee, C. Park, and S. Min, “A Flexible Tradeoff between Code Size and WCET Employing Dual Instruction Set Processors,” *International Workshop on Worst-Case Execution Time Analysis*, pp. 91-94 (July 2003).

[23] W. Zhao, P. Kulkarni, D. Whalley, C. Healy, F. Mueller, and G. Uh, “Tuning the WCET of Embedded Applications,” *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, (May 2004).

[24] F. Mueller and D. B. Whalley, “Avoiding Unconditional Jumps by Code Replication,” *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pp. 322-330 (June 1992).

[25] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).

[26] M. E. Benitez and J. W. Davidson, “A Portable Global Optimizer and Linker,” *Proceedings of the SIGPLAN ’88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[27] StarCore, Inc. and Atlanta, GA, *SC110 DSP Core Reference Manual*, 2001.

[28] StarCore, Inc. and Atlanta, GA, *SC100 Simulator Reference Manual*, 2001.