Improving WCET by Optimizing Worst-Case Paths

Wankang Zhao[1], William Kreahling[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

It is advantageous to perform compiler optimizations to lower the WCET of a task since tasks with lower WCETs are easier to schedule and more likely to meet their deadlines. Compiler writers in recent years have used profile information to detect the frequently executed paths in a program and there has been much effort to develop compiler optimizations to improve these paths in order to reduce average-case execution time. In this paper we describe our approach to reduce WCET by adapting and applying optimizations designed for frequent paths to the worst-case paths in an application. Our compiler uses feedback from our timing analyzer to detect the WCET paths through a function that will be subject to aggressive optimizations, reflect subsequent effects on the WCET of the paths due to these optimizations, and to also ensure that the worst-case path optimizations actually improve the WCET before committing to a code size increase. We evaluate a number of WC path optimizations and present results showing the decrease in WCET versus the increase in code size.

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. Simply measuring the execution time is not safe since it is difficult to determine input data that will produce the WCET. 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. An improvement in the WCET of a task may enable an embedded system to meet timing constraints that were previously infeasible. Improving the WCET of a task 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.

In an effort to improve the *average-case execution time* (ACET), compiler designers in recent years have used profile data to determine the frequently executed paths in a program and have developed compiler optimizations to improve the execution time of these paths [1]. Sometimes the optimizations performed on these frequent or hot paths may be at expense of the other paths in the function. Unfortunately, frequent paths optimizations are not guaranteed to reduce the WCET of an application since the most frequently executed paths may not be the WC paths.

In this paper we describe an approach for improving the WCET of an application by performing path optimizations on the WC paths of a function. We have integrated a timing analyzer with a compiler where the WCET of the application and the current function can be calculated on demand. The timing analyzer supplies the compiler with the WC path information and the compiler applies optimizations on the WC path that have been traditionally performed on frequent paths. After each code-improving transformation is performed on the WC path, the timing analyzer is invoked and up-to-date WCET path information is obtained in case the WC path has changed. When the WC path optimization increases code size, the timing analyzer is invoked to ensure that the WCET was reduced. If not, then the WC path transformation is discarded.

The remainder of the paper has the following organization. First, we outline related work in the areas of path optimization and improving WCET. Second, we present the experimental environment in which this research was accomplished. Third, we describe the WC path optimizations that we implemented within our compiler to automatically improve the WCET of the WC paths in a function. Fourth, we give results for a number of applications indicating the WCET improvement by applying these WC path optimizations. Finally, we give the conclusions for the paper.

2. Related Work

There has been a significant amount of work over the past couple of decades on developing optimizations to improve the performance of frequently executed paths. Each technique involves detecting the frequently executed path, distinguishing the frequent path using code duplication, and applying a variety of other code-improving transformations in an attempt to improve the frequent path, often at the expense of less frequently executed paths.

Much of this work was inspired by the goal of increasing the level of instruction-level parallelism in processors that can simultaneously issue multiple instructions. Some of the early work in this area involves a technique called trace scheduling, where long traces of the frequent path are obtained via loop unrolling and the trace is compacted into VLIW instructions [2]. A related technique that was later developed was called superblock formation and scheduling [3]. This approach differs in that tail duplication is used to make the trace have only a single entry point, which makes trace compaction simpler and more effective, though typically at the expense of an additional increase in code size as compared to trace scheduling.

Path optimizations have also been used to improve code for single issue processors. This includes techniques to avoid the execution of unconditional jumps [4] and conditional branches [5] and to perform partial dead code elimination [6] and partial redundancy elimination [7].

While there has been much work on developing compiler optimizations to reduce ACET and, to a lesser extent, to reduce space and power consumption, there has been relatively 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 [8]. 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 [9]. 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 [10]. 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 compiler optimizations to reduce the WCET of an application on a single instruction set processor. A genetic algorithm has been used to search for an effective optimization phase sequence that best reduces WCET for an application [11]. This search uses standard compiler optimizations, whereas we have developed optimizations that are driven by WCET path information. Finally, a WCET code positioning algorithm has been developed to find an ordering of the basic blocks in a function that attempts to minimize the number of unconditional jumps and taken conditional branches for transitions between basic blocks that can affect the WCET [12]. The optimizations we describe in this paper perform transformations to reduce WCET using code duplication and modification rather than just reordering blocks.

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, called VISTA (Vpo Interactive System for Tuning Applications), where a compiler can obtain WCET information from a timing analyzer upon demand [11]. 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 [13], which allows accurate WCETs to be obtained from a timing analyzer. One aspect of this system that we extensively used when applying WC path optimizations is the ability to discard previously applied transformations when the code size was increased without improving the WCET. This feature is accomplished by keeping a linked list of the transformations and their associated changes, discarding the current state of the program representation, reading in the intermediate representation for the current function, and applying the changes to the point of the last transformation that we wish to retain.

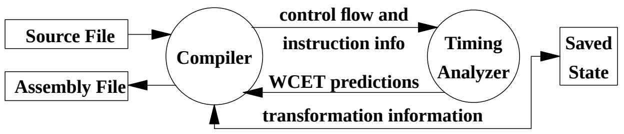


Figure 1: WCET Aware Compilation Process

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 [14, 15], a pipeline [16, 15], and a data cache [17, 18]. In addition, it can automatically calculate the maximum number of iterations of many loops, including those involving nonrectangular loop nests [19, 20]. Finally, the timing analyzer can also detect many constraints on branches that restrict the set of paths that can be taken in a program [21, 22].

Our timing analyzer uses a path-based approach, as opposed to using integer linear programming (ILP) [23, 24, 25] or symbolic execution [26, 27]. The result of ILP analysis is a single WCET prediction for the entire task, which means that it does not provide the detailed path information needed by a compiler to perform optimizations to improve WCET. The analysis time for symbolic execution is proportional to the WC number of instructions that would be executed. Thus, symbolic execution analysis can be prohibitively slow, which is not ideal for interfacing with a compiler. In contrast, our path-based approach both provides the WCET path information needed by a compiler to perform WCET path optimizations and performs the timing analysis very quickly [22].

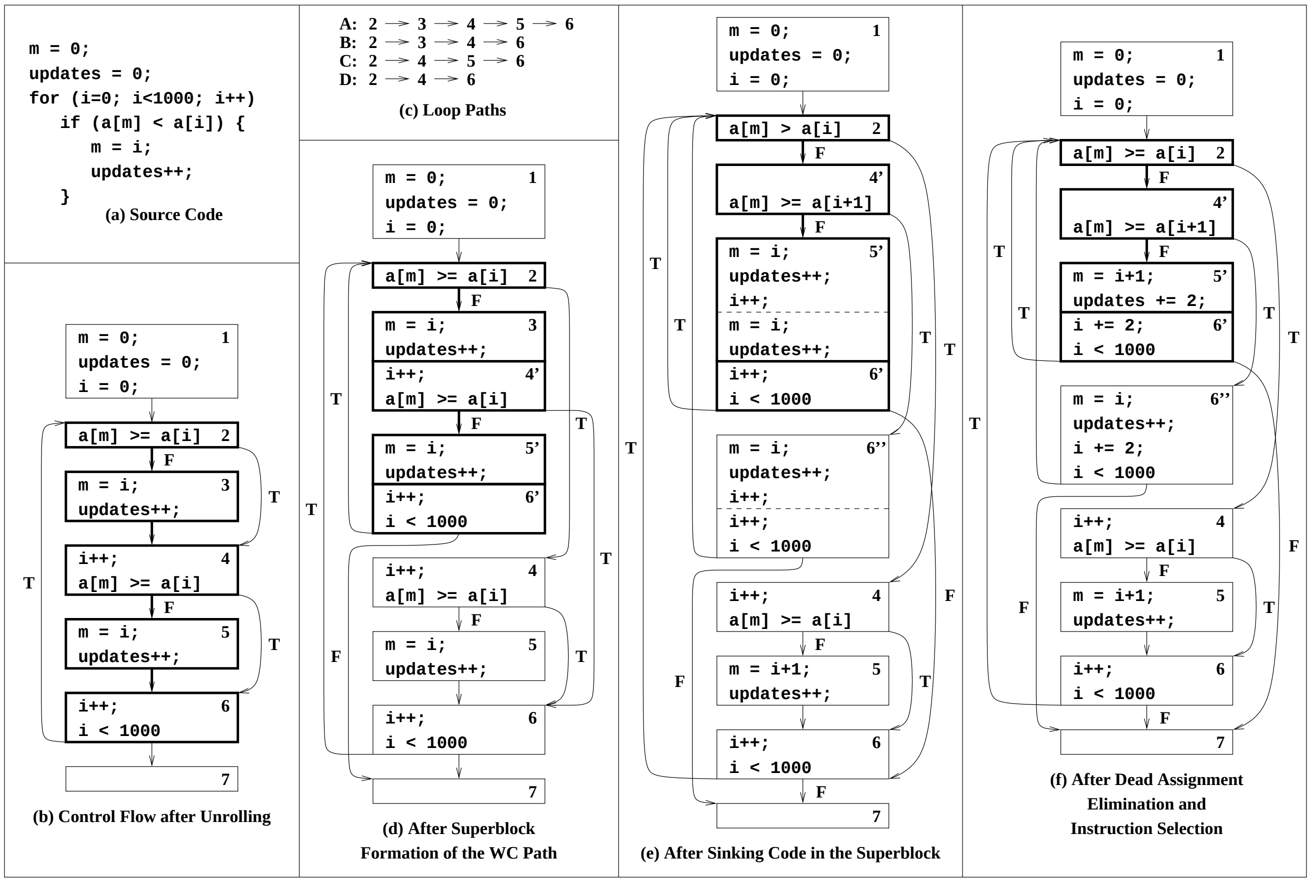
We have ported both the VPO compiler and our timing analyzer to the StarCore SC100 processor [28]. This processor has neither a memory hierarchy (no caches or virtual memory system) nor an OS, which facilitates obtaining tight WCET predictions [11]. It has no architectural support for floating-point operations since it is a digital signal processor that is instead designed 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 their 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. 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. 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.

4. WC Path Optimizations

Compilers that attempt to apply path optimizations typically identify the frequent paths within a program. Often optimizations applied for that path may be at the expense of less frequently executed paths. However, there is no guarantee that the WC path will be the same as the frequent path. For instance, consider Figure 2(a), which shows the source code for finding the index of the element for the maximum value in an array and the number of times that the index for the maximum element was updated. Figure 2(b) shows the corresponding control flow after unrolling the loop by a factor of two so that the loop overhead (compares and branches of the loop variable i) can be reduced. The WC path (blocks and transitions) is depicted in bold. Note that loop unrolling and all other optimizations are performed at a low level by the compiler backend to be able to assess the impact on both the WCET and code size. While the code in this 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 have been applied to allow more accurate timing predictions. The conditions have been reversed in the control flow to represent the condition tested by a conditional branch at the assembly level.

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 loop entry block (the loop header) and is terminated by a block that has a transition back to the header (the back edge) or outside the loop. A function path starts with the function entry block 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



```
m = 0;
updates = 0;
for (i=0; i<1000; i++)
   if (a[m] < a[i]) {
      m = i;
      updates++;
   }
```

```
A: 2 -> 3 -> 4 -> 5 -> 6
B: 2 -> 3 -> 4 -> 6
C: 2 -> 4 -> 5 -> 6
D: 2 -> 4 -> 6
```

(a) Source Code
(b) Control Flow after Unrolling
(c) Loop Paths
(d) After Superblock Formation of the WC Path
(e) After Sinking Code in the Superblock
(f) After Dead Assignment Elimination and Instruction Selection
T
F

Figure 2: Example Illustrating WCET Superblock Formation and Associated Optimizations

calculates the WCET path information on the original positioned code, then changing the order of the basic blocks due to performing WC path optimizations may result in unanticipated increases in the WCET for other paths since previously contiguous edges may become non-contiguous and be assessed a transfer of control penalty. We decided instead to treat the basic blocks as being initially unpositioned so that the current layout of the basic blocks does not bias the selection of the WC path. We accomplish this by actually modifying 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). Unnecessary jumps are later deleted so that an accurate WCET may be obtained to determine if the transformations are beneficial.

Figure 2(c) enumerates the four different paths through the loop. Transfer of control penalties are initially assessed between each basic block and path A is the current WC path in the loop due to it containing the most instructions. However, when the array contains random values, path D would likely be the most frequent path executed since not finding a new maximum is the most likely outcome of each iteration. Thus, this example illustrates that the frequent path and the WC path may differ.

We attempt WC path optimizations on the WC path in the innermost loops of a function or at the outer level if the function has no loops. Once the WC path is identified, we attempt superblock formation on that path. This means that we duplicate code so that the path is only entered at the header of the loop. Consider Figure 2(d), where a superblock (2→3→4'→5'→6') representing path A now is only entered at block 2. Blocks 4', 5', and 6' are duplicates of blocks 4, 5, and 6, respectively. Note that there are still multiple exits from this superblock, but there is only a single entry point.

Distinguishing the WC path may enable other compiler optimizations. In Figure 2(d), blocks 3 and 4' and blocks 5’ and 6’ are merged together. Removing joins (incoming transitions) from the WC path may enable some optimizations by itself.

When it is benefi cial, the compiler also sinks an instruction further down in the WC path. The two assignments in block 3 of Figure 2(d) and the increment of `i` from block 4’ in Figure 2(d) are sunk after the fallthrough transition of block 4’ into the top portion of block 5’ in Figure 2(e). Likewise, we have to duplicate these assignments after the taken transition of block 4’, which are assigned to the top portion of block 6’’. Due to the high cost of SC100 transfers of control, we continue to duplicate code until another transfer of control is encountered when sinking assignments off the WC path, as shown by the duplicated code in the bottom portion of block 6’’. This additional code duplication avoids introducing an extra unconditional jump at the end of block 6’’, which decreases the WCET of the path containing that block.

Initially it may appear that there is no benefi t from performing code sinking. Figure 2(f) shows the updated code after performing dead assignment elimination, instruction selection, and common subexpression elimination. The fi rst assignment to `m` in block 5’ of Figure 2(e) is now dead since its value is never used and this assignment is deleted in Figure 2(f). Likewise, the multiple increments to the *updates* variable in block 5’ of Figure 2(e) are combined into a single instruction in block 5’ of Figure 2(f). In addition, the two pair of increments of `i` in blocks 5’ and 6’ and in block 6’’ are combined into single increments in blocks 6’ and 6’’. Finally, the movement of the "`i++;`" statement past the assignment "`m = i;`" statement in block 5’ causes the source of that statement to be modifi ed. Other optimizations are also reapplied that can exploit the superblock control fbw with its single entry point. These optimizations include constant propagation, copy propagation, and strength reduction.

Figure 3 more clearly illustrates the WC superblock formation process. Figure 3(a) depicts the original control fbw. The blocks and transitions that are in bold indicate the WC path, 2→3→5→6→8, through the loop. We start at the beginning of the WC path and duplicate code up to the last point where other paths have an entry point (join block) into the WC path. Figure 3(b) shows the control fbw after duplicating code along the WC path. At this point there is only a single entry point in the WC path, which is the loop header at block 2. The algorithm also makes the blocks within the WC path contiguous, which eliminates transfers of control within the superblock. After superblock formation, the compiler also attempts other code improving transformations that may exploit the new control fbw and afterwards invokes the timing analyzer to obtain the WCET.

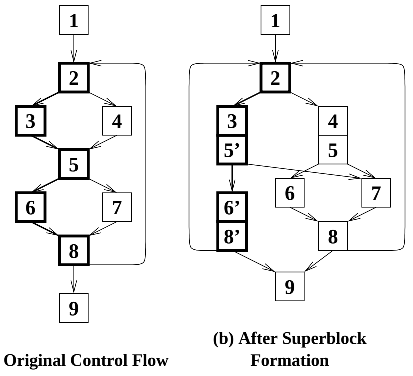


Figure 3: Example Illustrating Superblock Formation

Some transformations, such as distinguishing the WC path through WC superblock formation and code sinking, can increase the number of instructions in the function. Since our target application area is for embedded systems, we would prefer not to increase code size unless there was a corresponding benefi t obtained by decreasing the WCET. As mentioned previously, we have the ability in VISTA to discard previously applied transformations. Thus, the compiler invokes the timing analyzer to obtain the WCET before and after applying each code size increasing transformation. If the transformation does not decrease the WCET, then we restore the state of the program representation prior to the point when the transformation was applied. Alternatively, a user could specify the ratio of the code size increase to the WCET decrease that he/she is willing to accept. Note that the timing analyzer returns the WCET of the entire program. By checking the program’s entire WCET, the compiler discards all code size increasing transformations where the WC path does not contribute to the overall WCET, even though the transformation may decrease the WCET of the loop or function in which the WC path resides.

This ability to discard previously applied transformations also allows our compiler to aggressively apply an optimization in case the the resulting transformation will be benefi cial. For instance, notice that the number of taken conditional branches, which result in a transfer of control penalty on the SC100, could be reduced in the WC path within a loop by duplicating this path. For instance, regardless of how the nonexit path 2→3→5’→6’→8’→ in Figure 3(b) is positioned, it would require at least one transfer of control since the last transition is back to block 2. Figure 4 shows the control-fbw graph of Figure 3(b) after duplicating the WC path. Now the path 2→3→5’→6’→8’→2’→3’→5’’→6’’→8’’→ in Figure 4 can potentially be traversed with only a single transfer of control. For our study, we attempted WC path duplication on each of the innermost loops of a function.

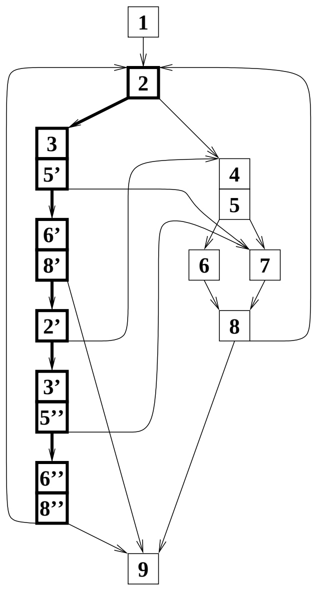


Figure 4: WC Path Duplication of Graph in Figure 3(b)

WC path duplication presents interesting challenges to the timing analyzer and the compiler since some acyclic paths, such as 2→...→8’’ in Figure 4, represent two iterations of the original loop and others, such as 2→4→...→8, represent a single iteration. We annotated the duplicated loop header, block 2’ in Figure 4, so that the timing analyzer counts an extra iteration for any path containing it. We also modified the compiler to retain the original number of loop iterations before WC path duplication and to halve the WCET of paths containing the duplicated loop header when performing code positioning [12].

Finally, we also investigated performing limited loop unrolling followed by superblock formation and associated other compiler optimizations to exploit the modified control flow. For our study we unrolled only the innermost loops of a function by a factor of two since we wished to limit the code size increase.[1] Figure 5(a) shows the control flow of Figure 3(a) after unrolling by an factor of two when the original loop had an even number of iterations. Figure 5(b) shows how our compiler uses a less conventional approach to perform loop unrolling by an unroll factor of two and still not require an extra copy of the loop body when the original number of loop iterations is odd. Each WC loop path (blocks and transitions) in these figures is again depicted in bold. Note that the WC loop path in Figure 5(b) starts at block 2’, the loop header, and ends at block 8. In both Figure 5(a) and 5(b) the compare and branch instructions in block 8 are eliminated, which will reduce both the ACET and WCET. However, the approach in Figure 5(b) does not result in any merged blocks, such as blocks 8 and 2’ in Figure 5(a), which may result in fewer other compiler optimizations being enabled. Figure 5(c) shows the code from Figure 5(a) after forming a superblock for the WC path. Likewise, a superblock could also be formed for the WC path in Figure 5(b). By contrasting Figure 4 with Figure 5(c), one can see that loop unrolling followed by superblock formation can result in a greater code size increase. While loop unrolling followed by superblock formation will likely result in lower WCETs, superblock formation followed by path duplication may be a more attractive alternative when the code size is constrained.

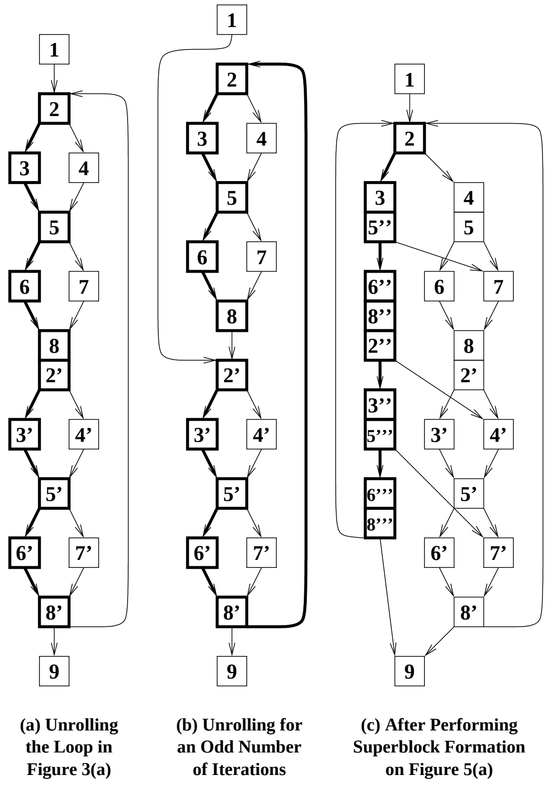

1
2
3
4
5
6
7
8
2’
3’
4’
5’
6’
7’
8’
9
5’’
6’’
8’’
2’’
3’’
5’’’
6’’’
8’’’
(a) Unrolling the Loop in Figure 3(a)
(b) Unrolling for an Odd Number of Iterations
(c) After Performing Superblock Formation on Figure 5(a)

Figure 5: Unrolling Followed by Superblock Formation

Prior to invoking the timing analyzer after performing each WC path optimization, we also perform instruction scheduling, WCET code positioning [12], insertion of noops to address data hazards (the SC100 has no pipeline interlocks), and WCET target alignment [12]. Much of the WCET improvement that was previously obtained

[1] Some approaches that perform unrolling of the superblock require a cleanup loop to handle exits from the superblock and this cleanup loop can be unstructured [29]. We could not use such an approach since our timing analyzer requires that all loops be structured for the analysis.

Table 1: Benchmarks Used in the Experiments

Program	Description
bubblesort	performs a bubble sort on 500 elements
findmax	finds the maximum element in a 1000 element array
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
sumposclr	sums positive values from two 1000 element arrays and sets negative values to zero
sym	tests if a 50x50 matrix is symmetric
unweight	converts an adjacency 100x100 matrix of a weighted graph to an unweighted graph

from WCET code positioning may now be achieved by superblock formation and WC path duplication due to the resulting contiguous layout of the blocks in the WC path.

5. Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of improving WCET by performing WC path optimizations. All of the optimizations described in the previous section were implemented in our compiler and the measurements were automatically obtained by applying these optimizations.

Table 1 shows the benchmarks we used for our experiments.[2] All of these benchmarks were selected since they have conditional constructs and have been used in a previous timing analysis studies.

Table 2 shows the accuracy of our timing analyzer. The measurements are taken after all optimizations have been applied except for those that are performed to improve the WC paths. We did include WCET target alignment [12], but did not include WCET code positioning [12] since we wish to show that the WC path optimizations often obtain much of the WCET code positioning benefit. The *observed cycles* were obtained by running the program executables through the SC100 simulator [30] using WCET input data. 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.[3] 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 SC100 generated code.

Table 2: Baseline Results

Program	Observed Cycles	WCET Cycles	WCET Ratio
bubblesort	7,248,033	7,499,047	**1.035**
findmax	19,997	20,002	**1.000**
keysearch	30,667	31,142	**1.015**
summidall	18,516	18,521	**1.000**
summinmax	23,009	23,015	**1.000**
sumnegpos	20,010	20,015	**1.000**
sumoddeven	22,525	22,549	**1.001**
sumposclr	31,013	31,018	**1.000**
sym	55,343	55,497	**1.003**
unweight	350,412	350,717	**1.001**
average	781,953	807,152	**1.006**

Table 3 shows the effect on WCET after performing superblock formation, WC path duplication, and WCET code positioning. Note these WC path optimizations are applied after all other conventional code-improving optimizations have been performed. For each of these optimizations, the transformation was not retained when the WCET was not improved. Thus, the code size was not increased unless the WCET was reduced. The results *after superblock formation* were obtained by applying superblock formation followed by a number of compiler optimizations to improve the code due to fewer joins in the superblock. Only three of the ten benchmarks improved. We sometimes found there are multiple paths in the benchmark that have the same WCET. In these cases improving one path does not reduce the WCET since the WCET for another path with the same WCET is not decreased. The WC path is also often already positioned with only fall through transitions, which occurs when `if-then` statements are used instead of `if-then-else` statements.

[2] The benchmark *findmax* contains the example code shown in Figure 2. However, we assigned the initial value for `i` in the `for` loop to be 1 instead of 0. Thus, when applying loop unrolling for this benchmark, the compiler uses the approach shown in Figure 5(b). We used an initial value of 0 in Figure 2 in order to simplify the example.

[3] The WCET input data had to be meticulously determined since the WCET paths were often difficult to detect manually due to control-flow penalties. We did not obtain *observed cycles* after applying WC path optimizations since this would typically require new WCET input data for each benchmark due to changes in the WCET paths.

Table 3: Results after Superblock Formation and WC Path Duplication

Program	After Superblock Formation				After WC Path Duplication				After WCET Positioning			
	WCET		Size	Time	WCET		Size	Time	WCET		Size	Time
	Cycles	Ratio	Ratio	Ratio	Cycles	Ratio	Ratio	Ratio	Cycles	Ratio	Ratio	Ratio
bubblesort	7,499,047	**1.000**	1.000	1.40	7,498,548	**1.000**	1.144	2.40	7,497,047	**1.000**	1.123	5.47
fi ndmax	20,002	**1.000**	1.000	1.29	20,002	**1.000**	1.000	2.14	18,010	**0.900**	1.655	5.43
keysearch	31,142	**1.000**	1.000	1.17	25,267	**0.811**	1.247	2.08	24,958	**0.801**	1.312	5.83
summidall	18,521	**1.000**	1.000	1.43	18,128	**0.979**	1.789	2.57	16,324	**0.881**	1.737	6.71
summinmax	23,015	**1.000**	1.000	1.33	23,015	**1.000**	1.000	2.33	20,021	**0.870**	1.067	5.22
sumnegpos	20,015	**1.000**	1.000	1.43	20,015	**1.000**	1.000	2.29	18,021	**0.900**	1.133	6.71
sumoddeven	16,547	**0.734**	1.038	1.63	16,547	**0.734**	1.392	2.38	16,546	**0.734**	1.000	4.50
sumposclr	30,019	**0.968**	1.420	1.27	30,019	**0.968**	1.951	2.18	26,024	**0.839**	2.222	6.09
sym	55,497	**1.000**	1.000	1.30	51,822	**0.934**	1.598	2.50	50,603	**0.912**	1.660	5.90
unweight	321,017	**0.915**	1.049	1.38	321,017	**0.915**	1.573	2.13	300,920	**0.858**	1.622	5.88
average	803,487	**0.962**	1.051	1.36	802,438	**0.934**	1.369	2.30	798,847	**0.870**	1.453	5.78

Changing the layout in this situation will not reduce the number of transfer of control penalties in the WC path. Finally, other optimizations often had no opportunity to be applied after superblock formation due to the path containing code for only a single iteration of the loop.

To obtain the results *after WC path duplication* we performed superblock formation followed by WC path duplication. If the WCET did not improve, then we discarded the transformations. In contrast to superblock formation alone, WC path duplication after superblock formation was more successful at reducing the WCET. First, assignments were often sunk across the duplicated loop header of the new WC path and other optimizations applied on the transformed code. Second, there was typically one less transfer of control after WC path duplication for every other original iteration. Eliminating a transfer of control is almost always benefi cial on the SC100.

The results *after WCET positioning* were obtained by performing superblock formation, WC path duplication, and WCET code positioning. Sometimes superblock formation and/or WC path duplication did not improve the WCET, but applying WCET code positioning in addition to these transformations resulted in an improvement. The combination of applying all three optimizations was over 4% more benefi cial on average than applying WCET code positioning alone. While superblock formation or WC path duplication did not always provide the best layout for the basic blocks, WCET code positioning could provide a better layout resulting in an additional improvement.

Table 4 shows the effect on WCET after unrolling innermost loops by a factor of two, superblock formation (as depicted in Figure 5), and WCET code positioning. As expected, loop unrolling reduced WCET. If typical input data was available for these benchmarks, then comparable benefi ts for ACET would be obtained. Five of the benchmarks improved *after superblock formation* was performed following loop unrolling. We found that eliminating one of the loop branches after unrolling caused other optimizations to be applied after superblock formation. WCET code positioning also improved the overall WCET for one half of the benchmarks beyond what could be accomplished by unrolling and superblock formation alone. The results in Table 4 show that the loop unrolling reduces WCET more than WC path duplication, but results in a greater increase in code size.

While the WCET is reduced by applying the WC path optimizations, there is an accompanying substantial code size increase, as shown shown in Tables 3 and 4. One must keep in mind that the benchmarks used in this study, like most timing analysis benchmarks, are quite small. Thus, the duplicated blocks from applying superblock formation, WC path duplication, and loop unrolling comprise a signifi cant percentage of the total code size. Performing these optimizations on larger applications should result in a smaller percentage code size increase.

The *time ratio*s in Tables 3 and 4 indicate the increase in compilation time from performing these optimizations. There were several factors that resulted in longer compilation times compared to those cited in a previous study [12]. First, the optimizations that we applied increased the number of basic blocks and paths in the program, which increased the timing analysis time and required additional invocations of the timing analyzer for WCET code positioning. Second, we had to perform required phases (fi xing the entry/exit of the function to address calling conventions and instruction scheduling to address the lack of pipeline interlocks) before invoking the timing analyzer. In contrast, WCET code positioning is performed after

Table 4: Results after Loop Unrolling and SuperBlock Formation

Program	After Loop Unrolling				After Superblock Formation				After WCET Positioning			
	WCET		Size	Time	WCET		Size	Time	WCET		Size	Time
	Cycles	**Ratio**	Ratio	Ratio	Cycles	**Ratio**	Ratio	Ratio	Cycles	**Ratio**	Ratio	Ratio
bubblesort	7,248,546	**0.967**	1.349	1.00	7,248,546	**0.967**	1.349	2.20	7,247,046	**0.966**	1.329	5.33
findmax	18,006	**0.900**	1.379	1.14	16,014	**0.801**	1.983	3.00	16,014	**0.801**	1.983	5.57
keysearch	28,767	**0.924**	1.435	1.08	24,767	**0.795**	1.242	1.75	24,767	**0.795**	1.242	3.75
summidall	16,520	**0.892**	1.386	1.29	16,520	**0.892**	1.386	2.57	15,077	**0.814**	2.105	9.29
summinmax	21,015	**0.913**	1.533	1.56	21,015	**0.913**	1.533	4.67	19,021	**0.826**	1.600	11.89
sumnegpos	17,015	**0.850**	1.400	1.14	17,015	**0.850**	1.400	5.57	16,021	**0.800**	1.533	20.00
sumoddeven	20,052	**0.889**	1.481	1.88	17,048	**0.756**	1.759	4.88	15,548	**0.690**	1.759	10.25
sumposclr	29,018	**0.936**	1.642	4.82	28,019	**0.903**	2.765	5.73	27,024	**0.871**	2.802	15.55
sym	50,597	**0.912**	1.546	1.10	50,597	**0.912**	1.546	1.90	49,372	**0.890**	1.546	4.20
unweight	330,716	**0.943**	1.561	1.25	311,017	**0.887**	2.098	2.88	311,017	**0.887**	2.098	6.50
average	778,025	**0.913**	1.471	1.63	775,056	**0.868**	1.706	3.51	774,091	**0.834**	1.800	9.23

these phases. We discarded these transformations after invoking the timing analyzer. We implemented this feature by reading in the intermediate file and reapplying the transformations up to the desired point in the compilation. The extra I/O to support this feature had a large impact on compilation time. The ability to discard previously applied transformations is not a feature that is available in most compilers.

As mentioned previously, a significant portion of the benefit from the WC path optimizations (superblock formation and WC path duplication) is obtained by the contiguous layout of the WC path. One should note that the WC path optimizations presented in this paper are computationally much less expensive than WCET code positioning, which requires an invocation of the timing analyzer after each time an edge is selected to be contiguous. Thus, the WCET code positioning requires many more invocations of the timing analyzer when it is performed. As shown in Tables 3 and 4, WCET code positioning has a much greater impact on compilation time.

6. Conclusions

In this paper we have described how the WCET of a program can be reduced by optimizing the WC paths. Compiler optimizations to improve the performance of paths typically use profile data to find the frequent paths in a program. In contrast, our compiler automatically uses feedback from our timing analyzer to detect the WCET paths through a function. We have shown that traditional frequent path optimizations can be applied to WC paths and improvements in the WCET can be obtained. In addition, we developed new optimizations, such as WC path duplication and constrained unrolling for an odd number of iterations, to improve WCET while minimizing code growth. We also found that it was critical to obtain feedback from the timing analyzer to ensure that each code size increasing transformation improves the WCET before allowing it to be committed.

During the course of this research, we realized that path optimizations applied on the WC path to reduce WCET will in general be less effective than reducing ACET when applied on the frequent path. One path within a loop may be executed much more frequently than other paths in the loop. In contrast, the WC path within a loop may be only slightly better than other paths. Performing optimizations on the WC path may quickly lead to another path having the greatest WCET, which can limit the benefit that can be obtained. However, we were able to show that reasonable WCET improvements can still be achieved by optimizing the WC paths of an application.

7. Acknowledgements

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

8. References

[1] T. Ball and J. Larus, "Using Paths to Measure, Explain, and Enhance Program Behavior," *Computer* **33**(7) pp. 57-65 (July 2000).

[2] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers* **30**(7) pp. 478-490 (July 1981).

[3] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab,

J. Holm, and D. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, pp. 229-248 (1993).

[4] 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).

[5] 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).

[6] R. Gupta, D. Berson, and J. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pp. 102-115 (1997).

[7] E. Morel and C. Renvoise, "Global Optimizations by Suppression of Partial Redundancies," *Communications of the ACM* **22**(2) pp. 96-103 (February 1979).

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

[9] 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).

[10] S. Lee, J. Lee, C. Park, and S. Min, "A Flexible Tradeoff between Code Size and WCET Using a Dual Instruction Set Processor," *International Workshop on Software and Compilers for Embedded Systems*, pp. 244-258 (September 2004).

[11] 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).

[12] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "WCET Code Positioning," *Proceedings of the IEEE Real-Time Systems Symposium*, (December 2004).

[13] 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).

[14] 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).

[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, 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).

[17] 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).

[18] 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).

[19] 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).

[20] 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).

[21] 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).

[22] 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).

[23] 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).

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

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

[26] 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).

[27] 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).

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

[29] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Third Edition,* Morgan Kaufmann, San Francisco, CA (2002).

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