Improving WCET by Applying Worst-Case Path Optimizations *

Wankang Zhao
Computer Science Department, Florida State University
wankzhao@cs.fsu.edu

William Kreahling
Department of Mathematics and Computer Science, Western Carolina University
wkreahling@email.wcu.edu

David Whalley
Computer Science Department, Florida State University
whalley@cs.fsu.edu

Christopher Healy
Computer Science Department, Furman University
chris.healy@furman.edu

Frank Mueller
Computer Science Department, North Carolina State University
mueller@cs.ncsu.edu

Abstract

It is advantageous to perform compiler optimizations that attempt to lower the *worst-case execution time* (WCET) of an embedded application 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 considerable effort to develop compiler optimizations to improve these paths in order to reduce the *average-case execution time* (ACET). In this paper, we describe an approach to reduce the WCET by adapting and applying optimizations designed for frequent paths to the *worst-case* (WC) paths in an application. Instead of profiling to find the frequent paths, our WCET path optimization uses feedback from a timing analyzer to detect the WC paths in a function. Since these path-based optimizations may increase code size, the subsequent effects on the WCET due to these optimizations are measured to ensure that the worst-case path optimizations actually improve the WCET before committing to a code size increase. We evaluate these WC path

*A preliminary version of this paper entitled "Improving WCET by Optimizing Worst-Case Paths" appeared in the 2005 *Real-Time and Embedded Technology and Applications Symposium*.

optimizations and present results showing the decrease in WCET versus the increase in code size.

Keywords: WCET, Path-based Optimizations, Embedded Systems

1 Introduction

Embedded applications often have to meet timing 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 the execution time of an embedded application can always meet its deadline.

Embedded systems are often deployed on *digital signal processors* (DSPs). Kernels for DSP applications have been historically written and optimized by hand in assembly code to ensure high performance (Eyre and Bier, 1998). However, assembly code is less portable and harder to develop, debug, and maintain. Many embedded applications are now written in high-level programming languages, such as C, to simplify their development. In order for these applications to compete with the applications written in assembly, aggressive compiler optimizations are used to ensure high performance.

Traditional path-based compiler optimizations are used to reduce the *average case execution time* (ACET), i.e. the typical execution time along the frequently executed path. Since the frequently executed paths may not be the *worst-case* (WC) paths, we propose applying path optimizations on WC paths to reduce the WCET of a task. An improvement in the WCET may enable an embedded system to meet timing constraints that were previously infeasible.

WCET constraints can impact energy consumption as well. In order to conserve energy, one can determine the WC number of cycles required for a task and lower the clock rate and still meet the timing constraint. A lower clock rate allows a developer to use lower voltage and save additional energy, which is valuable for mobile applications. In contrast, conservative assumptions concerning WCET may require the processor to be run at a higher clock rate, which consumes more energy.

Many researchers use profiling to identify the frequently executed paths. However, we need to identify the WC paths in a function. For our experiments, we integrated a timing analyzer with a compiler so the WCET path information of the current function can be calculated on demand. Figure 1 shows how the compiler obtains the WCET information. The compiler sends information about the control flow and the current instructions that have been generated to the timing analyzer. Predictions regarding the WCET path information are sent back to the compiler from the timing

analyzer. We use the VPO (*very portable optimizer*) compiler (Benitez and Davidson, 1988; Benitez, 1994; Benitez and Davidson, 1994) for this research. To assess the impact of performing these optimizations on the WCET, we retargeted both VPO and our timing analyzer to the StarCore SC100 processor, a DSP used in embedded systems (Star Core, 2001b).

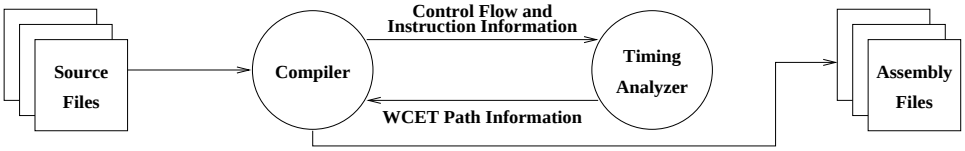

Figure 1: WCET Aware Compilation Process

The remainder of the paper is organized in the following fashion. In Section 2, we present work related to this research, including traditional path-based optimizations and prior work on compiler optimizations used to reduce WCET. The underlying architecture of this research is the StarCore SC100 processor. Therefore, in Sections 3 and 4, we introduce the StarCore SC100 processor and show how we retargeted our timing analyzer to it, respectively. In Section 5, we discuss how to integrate the timing analysis with the compiler. In Section 6, we describe the WC path optimizations used in this research. In Section 7, we illustrate how these path optimizations can enable other optimizations. In Section 8, we analyze the experimental results. In Section 9, we describe future work in this area and, in Section 10, we present our conclusions.

2 Related Work

Path-based compiler optimizations are typically used to reduce the execution time along the frequently executed paths to reduce the ACET of a program. This section summarizes the prior research on path-based compiler optimizations and WCET reduction techniques.

2.1 Prior Work on Path Optimizations

There has been a significant amount of work over the past few 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 and an increase in code size.

Much of this work was motivated 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 (Fisher, 1981). A related technique that was developed later is called superblock formation and scheduling (Hwu et al., 1993). This approach uses tail duplication to make a trace that has only a single entry point, which makes trace compaction simpler and more effective, though this typically comes at the expense of an additional increase in code size 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 (Mueller and Whalley, 1992) and conditional branches (Mueller and Whalley, 1995) and to perform partial dead code elimination and partial redundancy elimination (Gupta et al., 1997).

2.2 Other Prior Work on Reducing WCET

While there has been much work on developing compiler optimizations to reduce execution time and, to a lesser extent, to reduce space and energy consumption, there has been very little work on compiler optimizations to reduce WC performance. Marlowe and Masticola outlined a variety of standard compiler optimizations that could potentially affect the timing constraints of critical sections in a task (T. Marlowe, 1992). They proposed that some unsafe or dangerous transformations may even lengthen the execution time for some sets of inputs. These optimizations should be used with caution for real-time programs with timing constraints. However, no implementation was described in this paper. Hong and Gerber developed a programming language with timing constructs to specify the timing constraints and used a trace scheduling approach to improve code in critical sections of a program (Hong and Gerber, 1993). Based on these code-based timing constraints, they attempt to meet the WCET requirement for each critical section when performing code transformations. However, no empirical results were given since the implementation did not interface with a timing analyzer to serve as a guide for the optimizations or to evaluate the impact on reducing WCET. Both papers outlined strategies that attempt to move code outside of critical sections within an application that have been designated by a user to contain timing constraints. In contrast, most real-time systems use the WCET of an entire task to determine if a schedule can be met. Lee *et al.* used WCET information to choose how to generate code on a dual instruction set processor for the ARM and the Thumb (Lee et al., 2004). ARM code was generated for a selected subset of basic blocks that can impact the WCET while Thumb code was generated for the remaining blocks. In this way, they can reduce the WCET while minimizing code size. In contrast, we use

compiler optimizations to improve the WCET on a single instruction set processor.

2.3 Our Prior Work on Reducing WCET

Although compiler optimizations have been used to reduce the ACET for decades, compiler optimizations that attempt to reduce WCET comprise a relatively new research area. A genetic algorithm has been used to search for an effective optimization phase sequence that best reduces the WCET for an application (Zhao et al., 2004). The search space is the permutation of a sequence of traditional compiler optimization phases. After a sequence of optimization phases is applied, the WCET of the application is used to evaluate the efficiency for this particular ordering of optimization phases. However, this approach did not use WC path information from the timing analyzer to perform compiler optimizations.

A WCET code positioning optimization that uses WC path information has been developed to find an ordering of the basic blocks within a function to reduce WCET. This optimization attempts to minimize the number of unconditional jumps and taken conditional branches that can affect the WCET (Zhao et al., 2004). All basic blocks are treated as being initially unpositioned where every transition from one block to another has a transfer-of-control penalty. The algorithm selects one transition at a time in the *control flow graph* (CFG) and places two basic blocks, which are linked by the transition, next to each other. This operation eliminates the transfer-of-control penalty between two contiguous blocks. The algorithm then re-calculates the new WCET paths to guide the selection of the next edge. WCET code positioning typically does not increase the code size. In this paper, we present path-based compiler optimizations which reduce WCET using code duplication and modification rather than just reordering blocks.

3 The StarCore SC100 Architecture

The StarCore SC100, a DSP used in embedded system applications, is the architecture used in this research. There are no caches and no operating system in a SC100 system, which facilitates accurate WCET estimations. The SC100 contains a register file with 32 registers. 16 registers are used to store data (data registers) and 16 registers are used to store addresses (address registers). Each data register contains 40 bits and each address register contains 32 bits. The size of instructions can vary from one word (two bytes) to five words (ten bytes) depending upon the type of instruction, the addressing mode used, and the register number referenced.

The SC100 has a 5-stage pipeline. The five stages are: *Prefetch*, *Fetch*, *Dispatch*, *Address Genera-*

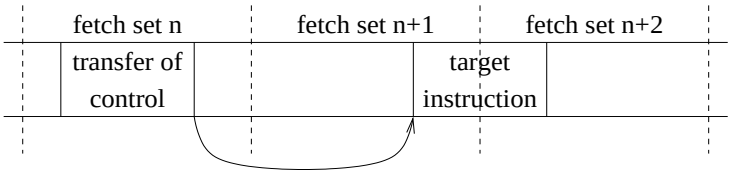

Figure 2: Example of a Misaligned Target Instruction

tion and *Execution*. The *prefetch* stage in the pipeline always fetches the next sequential instruction. Therefore, the SC100 processor incurs a pipeline delay whenever an instruction transfers control to a target that is not the next sequential instruction since the pipeline has to be flushed to fetch the target instruction. A transfer of control (taken branches, unconditional jumps, calls, and returns) results in a one to three cycle penalty depending on the type of the instruction, the addressing mode used, and if the transfer of control uses a delay slot. In this machine, if a conditional branch instruction is taken, then it takes three more cycles than if it was not taken. Unconditional jump instructions take two extra cycles if they use immediate values and take three extra cycles if they are PC-relative instructions.

Besides the pipeline delay, transfers of control on this machine also incur an extra delay if the target is misaligned. 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 in Figure 2. In this situation, the processor stalls for an additional cycle after the transfer of control (Star Core, 2001b).

In addition, there are no pipeline interlocks in this machine. It is the compiler's responsibility to insert no-op instructions to delay a subsequent instruction that uses the result of a preceding instruction when the result is not available in the pipeline. The SC100 architecture does not provide hardware support for floating-point data types, nor does it provide divide functionality for integer types (Star Core, 2001b).

4 Retargeting the Timing Analyzer to the SC100

WCET path-based optimizations need WC path information from a timing analyzer. In this section we describe a timing analyzer that was developed based on path analysis (Harmon et al., 1994; Arnold et al., 1994; Healy et al., 1995; Ko et al., 1996; Mueller, 1997; White et al., 1997; Ko et al., 1999; Healy et al., 1999; Healy and Whalley, 1999; White et al., 1999; Mueller, 2000; Healy et al.,

2000, 1999; Healy and Whalley, 2002) and how we retargeted it to the SC100 machine. This path analysis approach for predicting WCET involves the following issues:

- architecture modeling (pipeline and cache)
- detecting the maximum number of iterations of each loop
- detecting all possible paths and identifying infeasible paths
- analyzing each path to predict its WCET
- calculating the WCET for each loop and function
- predicting the WCET for the whole program based on a timing tree

An instruction's execution time can vary greatly depending on whether that instruction causes a cache hit or a cache miss. The timing analyzer starts by performing worst-case cache analysis (Arnold et al., 1994). The timing analyzer then integrates the cache analysis with pipeline analysis. Structural and data hazard pipeline information for each instruction is needed to calculate the execution time for a sequence of instructions. Cache misses and pipeline stalls are detected when inspecting each path (Healy et al., 1995).

When the WCET is needed for a benchmark, VPO generates the timing information for the benchmark and passes it to the timing analyzer. The timing analyzer uses the timing information as input to predict the WCET. Each instruction in the assembly has a counterpart in the timing information where the instruction type and the registers sets and uses are presented. The timing analyzer needs this information to detect pipeline hazards and find the worst-case paths. The timing information for the SC100 also contains the size information for each instruction, which is needed by the timing analyzer to detect the branch target mis-alignment penalties. In addition, the timing information contains the control flow information, such as the branches and loop, for the program.

The timing analyzer takes the timing information and produces the WCET path information (Arnold et al., 1994). Our timing analyzer calculates the WCET for all paths within each loop and the outer level of a function. A loop path consists of basic blocks and each loop path starts with the entry block (header) in the loop and is terminated by a block that has a transition back to the entry block (back edge) and/or a transition 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. The WCET for each path is calculated by the timing analyzer. This WC path information is used by the WC path optimizations.

Besides addressing architectural features, the timing analyzer also automatically calculates control-flow constraints to tighten the WCET. One type of constraint is determining the maximum number of iterations associated with each loop, including nonrectangular loops where the number of iterations of an inner loop depends on the value of an outer loop variable (Healy et al., 2000). Another type of constraint is called branch constraints. The timing analyzer uses such constraints to detect infeasible paths through the code and the frequency of how often a given path can be executed (Healy and Whalley, 2002). The timing analyzer uses the control-flow and constraint information, caching categorizations, and machine-dependent information (e.g. characteristics of the pipeline) to make its timing predictions.

The WCET of the whole program is obtained in a bottom-up fashion by following a timing tree, where the WCET for an inner loop (or called function) is calculated before determining the WCET for an outer loop (or calling function). Each function is treated as a loop with a single iteration. The WCET information for an inner loop (or called function) is used when it is encountered in an outer-level path.

Sometimes, the control flow within a loop has too many paths. For example, if there are 20 *if* statements inside a loop, there can be up to 2^{20} paths, which is beyond the capability of the timing analyzer to evaluate within a reasonable time. The timing analyzer was modified to partition the control flow of complex loops and functions into sections that are limited to a predefined number of paths (Ko et al., 1999). The timing tree is also updated to include each section as a direct descendant to the loop or function containing the section.

The architectural features of a machine, such as pipelining and caches, affect the WCET prediction of an application. We retargeted the timing analyzer to the SC100 processor to demonstrate that we could predict the WCET for an embedded machine. Since the SC100 does not have a memory hierarchy (no caches or virtual memory system), the timing analyzer was updated to treat all cache accesses as *hits* since instructions and data on the SC100 can in general be accessed in a single cycle from both ROM and RAM, respectively.

The timing analyzer was also modified to address the penalty for transfers of control. When calculating the WCET of a path, it has to be determined if each conditional branch in the path is taken or not since non-taken branches do not have this penalty. When there is a transfer-of-control penalty, the timing analyzer calculates the number of clock cycles of pipeline delay, which depends on the instruction type and whether there is an extra cycle due to a target misalignment penalty. Therefore, the size of each instruction is needed to find the SC100 fetch sets in order to calculate the target misalignment penalty.

In addition, we were able to obtain a simulator for the SC100 from StarCore (Star Core, 2001a). Many embedded processor simulators, in contrast to general-purpose processor simulators, can very closely estimate the actual number of cycles required for an application's execution. The SC100 simulator, which can simulate the SC100 executable and report an estimated number of execution cycles for a program, is used to verify the accuracy of the WCET timing analyzer. This simulator can report the size of each instruction as well, so we also used it to verify the code size obtained from the compiler.

5 Using WCET Information in the Compiler

We use an interactive compilation system called VISTA (VPO Interactive System for Tuning Applications) (Zhao et al., 2002; Kulkarni et al., 2003) to experiment with our WCET path optimizations. We retargeted VISTA to the SC100 architecture and integrated it with the SC100 timing analyzer. The VPO port of the SC100 compiler takes the source code and generates the assembly and the timing information, which is used by the timing analyzer to produce the WC path information. The compiler performs optimizations based on the WC path information and the user can see transformations of the program graphically step by step at the machine code level. After each optimization, the compiler automatically invokes the timing analyzer to update the WCET path information, which is used to guide the next optimization. This allows an embedded application to be developed at the source code level, but allows the developer to interactively tune the application at a low level to meet design constraints of the embedded system.

The WCET for the whole program is calculated in a bottom-up fashion based on a timing analysis tree, where each node is a loop or a function (see Section 4). The timing information of each node includes the execution time and the lists of the basic blocks along all the paths. The timing analyzer provides the timing information for all the nodes in the timing tree. However, the path optimizations are performed only on the innermost loops to limit the code size growth.

Several WC path optimizations are applied to transform the code after traditional optimizations have completed, but before some required phases such as *fix entry exit*, *instruction scheduling*, and *add noops*. *Fix entry exit* inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack. *Add noops* puts a noop between two dependent SC100 instructions when the first instruction produces a result that will not be available when the second instruction accesses it. This transformation is required since the SC100 has no pipeline interlocks. Some transformations, such as distinguishing the WC path through *superblock formation* and *code*

sinking, can increase the number of instructions in a function. Since we are performing optimizations for embedded systems, we would prefer not to increase code size unless there is a corresponding benefit gained from decreasing the WCET. In order to obtain the WCET before and after each path optimization, the required phases, *fix entry exit*, *instruction scheduling*, and *add noops* have to be applied. As mentioned previously, VISTA has the ability to roll back previously applied transformations. When the transformations need to be reversed, the current function, with all its data structures, is discarded and re-initialized. A fresh instance of the function is then read from the input file. The optimizations are re-applied up to the point before or after the WC path optimization was applied, depending on if the optimization reduces the WCET.

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. 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 the compiler to aggressively apply an optimization in case the resulting transformation will be beneficial.

6 WC Path Optimizations

Traditional frequent path-based optimizations are performed based upon the path frequency information gathered from profiling. In contrast, WC path optimizations need to be applied on the paths in a function that have the highest WCET. To obtain this information, we integrated the timing analyzer with the compiler. Therefore, path optimizations, such as *superblock formation*, can be applied along the worst-case paths. In this section, we introduce three WC path-based optimizations.

6.1 Superblock Formation

A superblock is a sequence of basic blocks in the *control-flow graph* (CFG) where the control can only enter at the top but there may be multiple exits. Each block within the superblock, except for the entry block, can have at most one predecessor. Although *superblock formation* can increase code size due to code duplication, it can also enable other optimizations to reduce the execution time along the superblock.

Figure 3 illustrates the WC *superblock formation* process. Figure 3(a) depicts the original control flow of a function. Assume that the timing analyzer indicates that the WC path through the loop is 2→3→5→6→8. Note that the blocks and transitions along the WC path are shown in bold font. We start at the beginning of the WC path and duplicate code from the point where other paths have an entry point (join block) into the WC path. In this example, block 5 is a join block. Figure 3(b) shows the control flow 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. Blocks 5', 6', and 8' are duplicates of blocks 5, 6, 8, respectively. Although block 5' forks into block 6' and block 7, there is no join block along the WC path. To eliminate transfer of control penalties within the superblock, the compiler makes the blocks within the WC path contiguous in memory, which eliminates transfers of control within the superblock. After superblock formation some blocks can be merged. For instance, blocks 3 and 5' and blocks 6' and 8' can both be merged into one block. The compiler then attempts other code improving transformations that may exploit the new control flow and afterwards invokes the timing analyzer to obtain the new WCET.

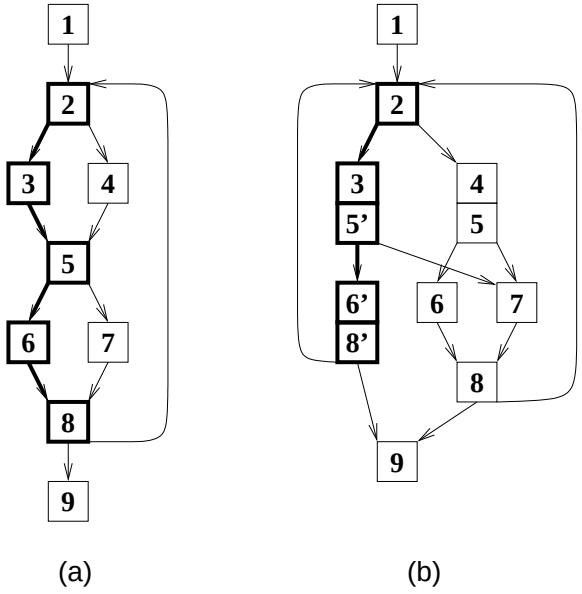

Figure 3: Example Illustrating Superblock Formation

6.2 Path Duplication

Figure 4 shows the control-flow graph from Figure 3(b) after *path duplication* to duplicate the WC path. The number of taken conditional branches, which result in a transfer of control penalty on the SC100, is 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. If *path duplication* duplicates the WC path once, one iteration of the worst-case path after duplication is equivalent to two iterations before the duplication. Now the path 2→3→5→6'→8'→ 2'→3'→5"→6"→8"→2 in Figure 4 can potentially be traversed with only a single transfer of control. In contrast, at least two transfers of control would be required before path duplication to execute the code that is equivalent to this path. In addition, WC path duplication forms one superblock consisting of code from two loop iterations which can enhance the opportunities for other optimizations.

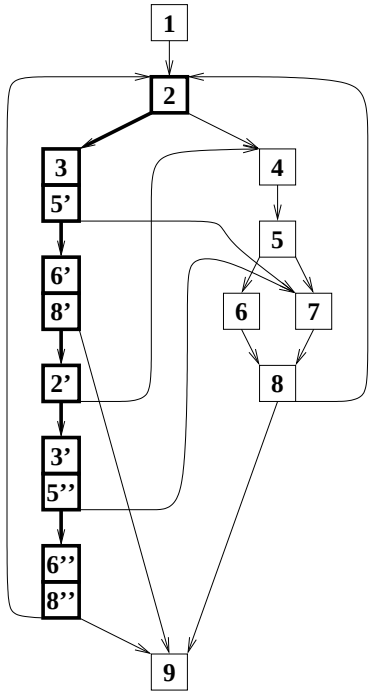

Figure 4: WC Path Duplication of Graph in Figure 3(b)

WC path duplication presents interesting challenges for the timing analyzer and the compiler since some acyclic paths, such as 2→3→...→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 count two original loop iterations for each path containing the duplicated loop header.

6.3 Loop Unrolling

Loop unrolling reduces the loop overhead by duplicating the whole loop body. It is different from *path duplication*, where only the WC path is duplicated. We investigate performing limited *loop unrolling* followed by superblock formation and associated other compiler optimizations to exploit the modified control flow. For this study, only the innermost loops of a function are unrolled by a factor of two since we wished to limit the code size increase. Some approaches that perform unrolling require a cleanup loop to handle exits from the superblock and this cleanup loop can be unstructured. We did not use such an approach since our timing analyzer requires that all loops be structured for the analysis and this approach would result in a larger code size increase.

Figure 5(a) shows the control flow from Figure 3(a) after unrolling by a 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 an odd number. 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, reducing 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) show superblock formation after loop unrolling. As illustrated in Figures 4 and 5, *path duplication* results in less code duplication than *loop unrolling*. However, *loop unrolling* can result in a greater reduction in WCET than *path duplication*.

7 Enabling Other Optimizations

Superblock formation, *path duplication*, and *loop unrolling* may enable other compiler optimizations. For instance, consider Figure 6(a), which shows the source code for a program that finds the index of the element for the maximum value in an array and counts the number of times that the index for the maximum element was updated. Figure 6(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. 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

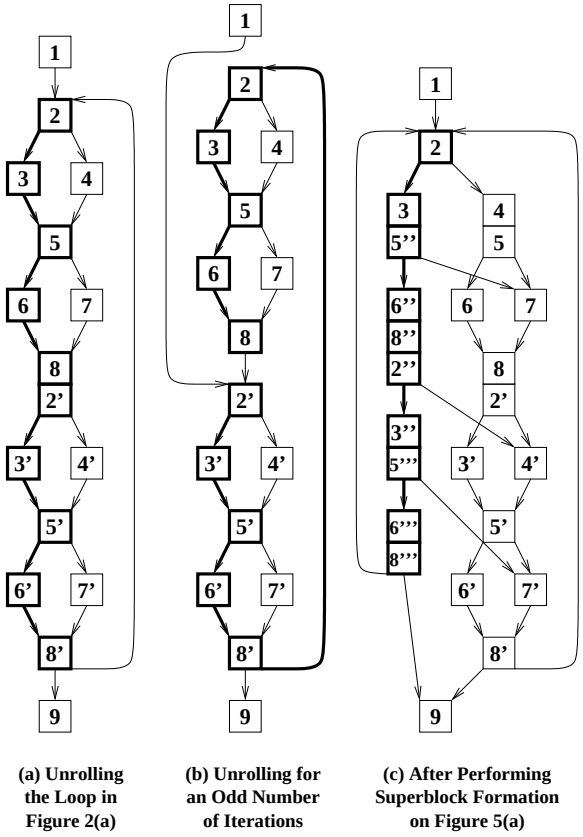

Figure 5: Unrolling Followed by Superblock Formation

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.

Figure 6(c) enumerates the four different paths through the loop. Transfer of control penalties are initially assessed between each pair of basic blocks. Path A is the current WC path in the loop because it contains 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. In this example, the frequent path and the WC path are different.

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 to limit the code size increase. The innermost loops and functions with no loops are the leaves of the timing tree which are often comprise most of the execution time. Once the WC path is identified, we attempt *superblock formation* on that path. This means that code is duplicated so that the path is only entered at the head of the loop. Consider Figure 6(d), where 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

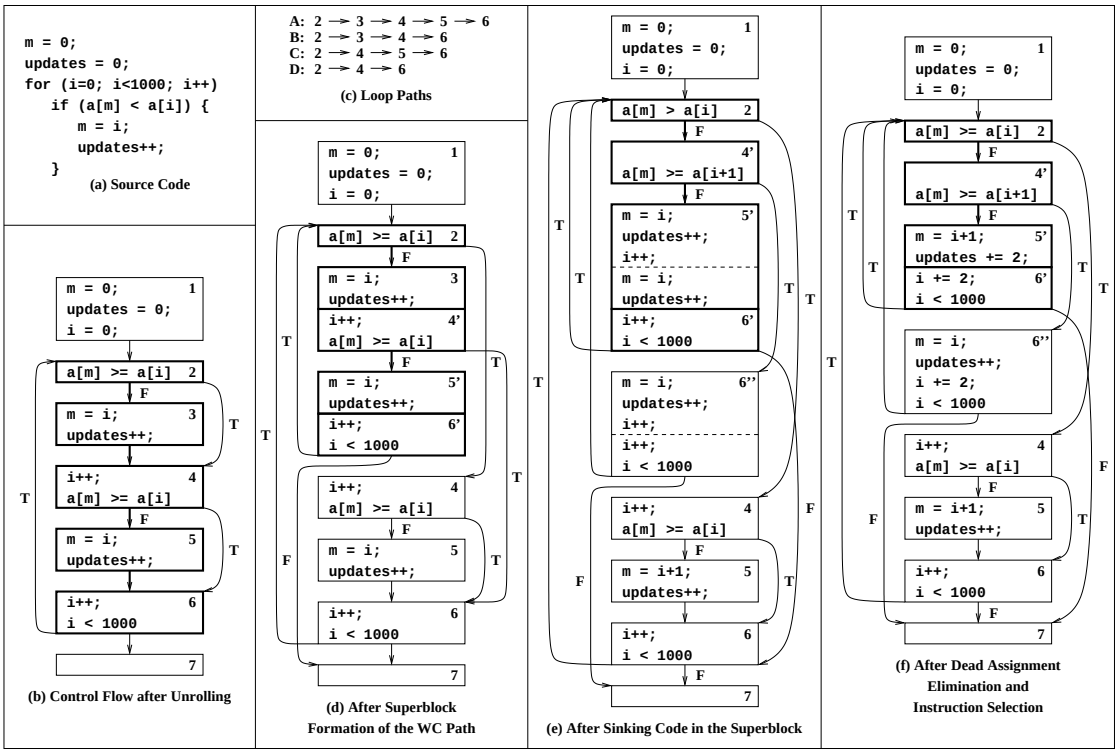

Figure 6: Illustrating WCET Superblock Formation and Associated Optimizations

path may enable other compiler optimizations. In Figure 6(d), blocks 3 and 4' and blocks 5' and 6' are merged together. Removing joins (incoming transitions) to the WC path may also enable other optimizations.

Path duplication is performed after *superblock formation* since *superblock formation* eliminates the *join* transitions along the WC paths. *Superblock formation* also makes it possible for an optimization called *code sinking* on instructions along the WC path to reduce the WCET. When it is beneficial, *coding sinking* moves instructions inside a block downward following the control-flow into its successor blocks. The instructions being pushed down can sometimes be merged with instructions in the successor block along the worst-case path. However, if a block has more than one successor, the moved instructions have to be duplicated for each successor, which potentially increase the code size. The two assignments in block 3 of Figure 6(d) and the increment of i from block 4' in Figure 6(d) are sunk after the fallthrough transition of block 4' into the top portion of block 5' in Figure 6(e). Likewise, these assignments have to be duplicated after the taken transition of block 4' in the top portion of block 6". Due to the high cost of SC100 transfers of control, code duplication is performed until another transfer of control is encountered when code sinking leads to assignments being removed 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 benefit from performing *code sinking*. Figure 6(f) shows the updated code after performing dead assignment elimination, instruction selection, and common subexpression elimination. The first assignment to m in block 5' of Figure 6(e) is now dead since its value is never used and this assignment is deleted in Figure 6(f). Likewise, the multiple increments to the *updates* variable in block 5' of Figure 6(e) are combined into a single instruction in block 5' of Figure 6(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 modified. Other optimizations, such as constant propagation, copy propagation, and strength reduction, are also re-applied to exploit the superblock.

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

8 Results

This section describes the results of a set of experiments to illustrate the accuracy of the SC100 timing analyzer and the effectiveness of improving the WCET by using WCET path optimizations. All of the optimizations described in the previous sections were implemented in our compiler and the measurements were automatically obtained after applying these optimizations. Table 1 shows the benchmarks and applications used to test the WCET reduction optimizations. These include benchmarks or programs used in previous studies by various groups (FSU, SNU, Uppsala) working on WCET timing analysis. These benchmarks in Table 1 were selected since they have conditional constructs, which means the WCET and ACET input data may not be the same.

The benchmark *findmax* contains the code similar to the example shown in Figure 6. For the example in Figure 6, the initial value of i in the *for* loop is 0, so the loop has an even number of the loop iterations, which simplifies the example since loop unrolling can use the approach shown in Figure 5(a). However, in the benchmark *findmax*, we assigned the initial value for i in the *for*

Table 1: Benchmarks Used in the Experiments

	Program	Description
Small	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
	sumnegpos	sums the negative, positive, and all elements of a 1000 integer vector
	sumoddeven	sums the odd and even 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
Larger	bitcnt	five different methods to do bit-count
	diskrep	train communication network to control low-level hardware equipments
	fft	128 point complex FFT
	fire	fire encoder
	sha	secure hash algorithm
	stringsearch	Pratt-Boyer-Moore string search

loop to be 1 instead of 0 since the first iteration of the loop is unnecessary. The loop has an odd number of iterations. Thus, when applying loop unrolling for this benchmark, the compiler uses the approach shown in Figure 5(b).

All input and output was performed by reading from and writing to global variables, respectively, to avoid having to estimate the WCET of performing actual I/O. If the input data for the original benchmark was from a file, then we modified the benchmark so that a global array is initialized with constants. Likewise, output is written to a global array.

In order to verify the accuracy of the worst-case timing analyzer, the SC100 simulator from StarCore is used to obtain the execution time driven by the WC input data. Table 2 shows the baseline measurements of the experiments. The measurements are taken after all optimizations have been applied except for those that are performed to improve the WC paths. *WCET cycles* are obtained from the timing analyzer and *observed cycles* are obtained from the SC100 simulator. *WCET cycles* should be larger than or equal to the *observed cycles* since the WCET is the upper bound for the execution time and it should never be underestimated. The WC input data has to be meticulously determined since the WC paths were often difficult to detect manually due to control-flow penalties. Therefore, these benchmarks are classified into two categories: *Small* and *Larger* benchmarks. The WC input data can be detected manually for *Small* benchmarks. Therefore, the

Table 2: The Baseline Code Size, Observed Cycles, and WCET

Category	Benchmarks	Code Size (bytes)	Lines of Source	Observed Cycles	WCET Cycles	**WCET Ratio**
Small	bubblesort	145	93	7,372,782	7,623,795	**1.034**
	findmax	58	21	19,997	20,002	**1.000**
	keysearch	186	537	30,667	31,142	**1.015**
	summidall	56	23	19,513	19,520	**1.000**
	summinmax	60	47	23,009	23,015	**1.000**
	sumnegpos	45	20	20,010	20,015	**1.000**
	sumoddeven	78	51	22,025	23,032	**1.046**
	sumposclr	81	35	31,013	31,018	**1.000**
	sym	97	40	55,343	55,497	**1.003**
	unweight	79	23	350,507	350,814	**1.001**
small average		89	89	794,487	819,785	**1.010**
Larger	bitcnt	354	170	39,616	55,620	**1.404**
	diskrep	388	500	9,957	12,494	**1.255**
	fft	631	220	73,766	73834	**1.001**
	fire	247	109	8,813	10,210	**1.159**
	sha	907	253	691,045	769,493	**1.114**
	stringsearch	333	237	147,508	194,509	**1.319**
larger average		477	248	161,784	186,027	**1.208**
overall average		234	149	557,223	582,126	**1.084**

WCET from the timing analyzer is close to the execution time obtained from the simulator. However, the WC input data is more difficult to manually detect for the *Larger* benchmarks. Therefore, the WCET from the timing analyzer may be much larger than the execution time obtained from the simulator for these larger benchmarks. This does not necessarily imply that the timing analyzer is inaccurate, but rather that the input data is not causing the execution of the WC paths. The *WCET ratios* show that these predictions are reasonably close for *Small* programs, but much larger on average than the *observed cycles* for the *larger* benchmarks. We did not obtain the *observed cycles* after WCET path optimizations since this would require new WCET input data due to changes in the WCET paths. Table 2 also shows the instruction code size and the lines of source code for these benchmarks. The instruction code size of the *Larger* benchmarks is no less than 250 bytes while the code size is under 200 bytes for each of *Small* benchmarks.

Two sets of experiments were performed to assess the effectiveness of applying WC path optimizations. The first experiment invokes superblock formation along the worst-case path. Path duplication is then performed to duplicate the superblock to reduce the number of transfers of control along the worst-case path. The second experiment applies loop unrolling on the innermost loop. Superblock formation is then performed to create a superblock along the worst-case path.

After each set of WC path optimizations, other optimizations, such as code sinking, merging basic blocks, dead assignment elimination, and instruction selection, are invoked to reduce the execution time along these worst-case paths. Finally, WCET code positioning is invoked to further reduce the WCET (Zhao et al., 2004).

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 the simplified control flow in the superblock. Only five of the ten *Small* benchmarks and five of the six *Larger* benchmarks improved. There are several reasons why there is no improvement on WCET after *superblock formation*. Sometimes, 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. There is no opportunity to change the layout in this situation to 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.

The results after *WC path duplication* shown in the middle portion of Table 3 were obtained by performing *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 could be applied on the transformed code. Second, there was typically one transfer of control eliminated after WC *path duplication* for every other original iteration. Eliminating a transfer of control is almost always beneficial on the SC100.

The results after *WCET positioning* for the final column in Table 3 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 beneficial on average than applying WCET *code*

Table 3: Results after Superblock Formation and WC Path Duplication

Program	Superblock Formation		Path Duplication		Code Positioning		
	WCET	Size	**WCET**	Size	WCET		Size
	Ratio	Ratio	**Ratio**	Ratio	Cycles	**Ratio**	Ratio
bubblesort	**0.984**	1.007	**0.951**	1.586	7,248,051	**0.951**	1.586
findmax	**1.000**	1.000	**1.000**	1.000	18,010	**0.900**	1.655
keysearch	**1.000**	1.000	**0.811**	1.247	24,958	**0.801**	1.312
summidall	**0.949**	1.018	**0.929**	1.821	16,325	**0.836**	1.804
summinmax	**1.000**	1.000	**1.000**	1.000	20,021	**0.870**	1.067
sumnegpos	**1.000**	1.000	**1.000**	1.000	18,021	**0.900**	1.133
sumoddeven	**0.718**	1.051	**0.718**	1.410	16,546	**0.718**	1.013
sumposclr	**0.968**	1.420	**0.968**	1.951	26,024	**0.839**	2.222
sym	**1.000**	1.000	**0.934**	1.598	50,603	**0.912**	1.660
unweight	**0.915**	1.089	**0.915**	1.633	300,920	**0.858**	1.684
small average	**0.953**	1.058	**0.923**	1.425	773,948	**0.859**	1.514
bitcnt	**0.998**	1.003	**0.910**	1.164	49,023	**0.881**	1.161
diskrep	**1.000**	1.000	**1.000**	1.000	11,905	**0.953**	1.021
fft	**0.994**	0.998	**0.961**	1.580	70,891	**0.960**	1.583
fire	**0.948**	1.105	**0.934**	1.765	9,395	**0.920**	1.789
sha	**0.987**	1.000	**0.953**	1.218	733,450	**0.953**	1.225
stringsearch	**0.998**	1.039	**0.894**	1.447	167,893	**0.863**	1.432
larger average	**0.987**	1.024	**0.942**	1.362	173,760	**0.922**	1.369
overall average	**0.966**	1.046	**0.930**	1.401	548,877	**0.882**	1.459

positioning alone. While *superblock formation* or WC *path duplication* did not always provide the best layout for the basic blocks, WCET *code positioning* in the final stage usually results in a better layout with an additional improvement. Results from applying WCET code positioning without the WC path optimizations described in this paper are described elsewhere (Zhao et al., 2005).

The effect on ACET after applying *superblock formation*, *path duplication*, and *code positioning* to improve WCET is shown in Table 4. After *superblock formation*, the average ACET is reduced by 3.2%. After *path duplication*, the average ACET is reduced by 6.9%. The average ACET of these benchmarks is also reduced after *code positioning*. The benefit to WC paths will help ACET if the random input data drives the WC path. Sometimes, the WC path optimization is not applied for some benchmarks shown in Table 4 if there is no improvement on WCET. However, it also causes no improvement on ACET. Overall, the improvement on ACET is comparable to the WCET improvement.

Table 5 shows experimental results for the second experiment. First, the effect on WCET and code size after unrolling innermost loops by a factor of two is shown. Second, the results after *superblock formation* (as depicted in Figure 5) are depicted. Finally, the results after WCET code positioning are given. As expected, loop unrolling reduced WCET for all benchmarks. If typical

Table 4: ACET Results after Superblock Formation and WC Path Duplication

	Default	Superblock Formation		Path Duplication		Code Positioning	
Program	ACET	ACET	Ratio	ACET	Ratio	ACET	Ratio
bubblesort	5,086,177	5,025,915	0.988	4,891,925	0.962	4,889,039	0.961
findmax	19,991	19,991	1.000	19,991	1.000	17,006	0.851
keysearch	11,067	11,067	1.000	9,173	0.829	9,016	0.815
summidall	19,511	18,514	0.949	17,913	0.918	16,122	0.826
summinmax	23,009	23,009	1.000	23,009	1.000	20,018	0.870
sumnegpos	18,032	18,032	1.000	18,032	1.000	15,042	0.834
sumoddeven	14,783	11,098	0.751	11,098	0.751	11,097	0.751
sumposclr	28,469	27,255	0.957	26,416	0.928	24,795	0.871
sym	107	107	1.000	107	1.000	107	1.000
unweight	340,577	315,517	0.926	305,510	0.897	290,939	0.854
small average	556,172	547,051	0.957	532,317	0.928	529,318	0.863
bitcnt	39,616	39,517	0.998	37,215	0.939	36,015	0.909
diskrep	9,957	9,955	1.000	9,955	1.000	9,566	0.961
fft	73,766	73,318	0.994	70,855	0.961	70,802	0.960
fire	8,813	8,280	0.940	8,151	0.925	8,067	0.915
sha	691,045	683,046	0.988	648,892	0.939	648,896	0.939
stringsearch	147,508	147,339	0.999	125,222	0.849	125,057	0.848
larger average	161,784	160,243	0.986	150,048	0.935	149,734	0.922
overall average	408,277	401,998	0.968	388,967	0.931	386,974	0.885

input data were available for these benchmarks, then comparable benefits for ACET would be obtained. Six out of ten *Small* benchmarks and five out of six *Larger* benchmarks improved after *superblock formation* was performed following loop unrolling. We found that eliminating one of the loop branches after unrolling enabled other optimizations to be applied after *superblock formation*. WCET *code positioning* also improved the overall WCET for half of the benchmarks, beyond what could be accomplished by unrolling and *superblock formation* alone. The results in Table 5 show that loop unrolling reduces WCET more than WC *path duplication*.

While the WCET is reduced by applying WC path optimizations, there is an accompanying substantial code size increase, as shown shown in Tables 3 and 5. For small benchmarks, the duplicated blocks from applying *superblock formation*, WC *path duplication*, and *loop unrolling* comprise a significant percentage of the total code size. Performing these optimizations on larger applications results in smaller code size increases. We anticipate applications that are even larger will exhibit progressively smaller code size increases since the paths on which the transformations are performed will represent a smaller percentage of the total code size. As expected, *loop unrolling* followed by *superblock formation* results in a greater code size increase than *superblock formation* followed by WC *path duplication*. The type of WC path optimization that should be applied depends

Table 5: Results after Loop Unrolling and Superblock Formation

Program	Loop Unrolling		Superblock Formation		Code Positioning		
	WCET Ratio	Size Ratio	**WCET Ratio**	Size Ratio	WCET Cycles	**WCET Ratio**	Size Ratio
bubblesort	**0.951**	1.359	**0.935**	1.724	7,122,301	**0.934**	1.717
findmax	**0.900**	1.379	**0.801**	1.983	16,014	**0.801**	1.983
keysearch	**0.924**	1.435	**0.795**	1.242	24,767	**0.795**	1.242
summidall	**0.846**	1.411	**0.846**	1.411	14,728	**0.755**	2.143
summinmax	**0.913**	1.533	**0.913**	1.533	19,021	**0.826**	1.600
sumnegpos	**0.850**	1.400	**0.850**	1.400	16,021	**0.800**	1.533
sumoddeven	**0.871**	1.500	**0.740**	1.782	15,548	**0.675**	1.782
sumposclr	**0.936**	1.642	**0.903**	2.765	27,024	**0.871**	2.802
sym	**0.912**	1.546	**0.912**	1.546	49,372	**0.890**	1.546
unweight	**0.943**	1.620	**0.887**	2.177	311,017	**0.887**	2.177
small average	**0.904**	1.483	**0.858**	1.756	761,581	**0.823**	1.853
bitcnt	**0.912**	1.113	**0.885**	1.121	47,720	**0.858**	1.113
diskrep	**0.968**	1.242	**0.968**	1.242	11,713	**0.937**	1.258
fft	**0.995**	1.203	**0.978**	1.192	72,178	**0.978**	1.197
fire	**0.969**	1.255	**0.901**	1.696	9,184	**0.900**	1.704
sha	**0.953**	1.092	**0.926**	1.086	712,467	**0.926**	1.093
stringsearch	**0.956**	1.330	**0.949**	1.417	179,578	**0.923**	1.441
larger average	**0.959**	1.206	**0.935**	1.293	172,140	**0.920**	1.301
overall average	**0.925**	1.379	**0.887**	1.582	540,541	**0.860**	1.646

on the timing constraints and code size limitation that should be met.

The effect on ACET after WCET path optimization for the second experiment is shown in Table 6. For the benchmarks in Table 6, *loop unrolling* reduces both ACET and WCET since it duplicates all paths. WC superblock formation and WCET code positioning reduce ACET when the input data causes the program to traverse the WC path. The average ACET is reduced by 7.5% after *loop unrolling*, 10.8% after WC *superblock formation*, and 13.4% after WCET *code positioning*. As in the first experiment, the average benefit on ACET is slightly less than the average benefit on WCET since WC paths are targeted during WC path optimizations.

The *time ratio* columns in Tables 7 and 8 indicate the compilation overhead from performing these optimizations. Most of this overhead is due to repeated calls to the timing analyzer. There were several factors that resulted in longer compilation times. First, the applied optimizations increased the number of basic blocks and paths in the program, which increased time for needed for timing analysis and required additional invocations of the timing analyzer for WCET code positioning. Second, we had to perform required phases (fixing 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. These transformations were discarded after invoking the timing analyzer by reading

Table 6: ACET Results after Loop Unrolling and Superblock Formation

	Default	Loop Unrolling		Superblock Formation		Code Positioning	
Program	ACET	ACET	Ratio	ACET	Ratio	ACET	Ratio
bubblesort	5,086,177	4,721,255	0.928	4,703,142	0.925	4,699,933	0.924
findmax	19,991	17,498	0.875	16,005	0.801	16,005	0.801
keysearch	11,067	10,353	0.935	8,913	0.805	8,913	0.805
summidall	19,511	16,511	0.846	16,511	0.846	14,746	0.756
summinmax	23,009	20,509	0.891	20,509	0.891	19,018	0.827
sumnegpos	18,032	15,031	0.834	15,031	0.834	13,541	0.751
sumoddeven	14,783	13,442	0.909	11,431	0.773	10,426	0.705
sumposclr	28,469	25,969	0.912	25,636	0.900	24,544	0.862
sym	107	105	0.981	105	0.981	102	0.953
unweight	340,577	315,480	0.926	300,366	0.882	300,366	0.882
small average	556,172	515,615	0.904	511,765	0.864	510,759	0.827
bitcnt	39,616	36,816	0.929	35,916	0.907	34,716	0.876
diskrep	9,957	9,527	0.957	9,525	0.957	9,358	0.940
fft	73,766	72,990	0.989	72,166	0.978	72,114	0.978
fire	8,813	8,413	0.955	7,796	0.885	7,785	0.883
sha	691,045	650,957	0.942	636,354	0.921	636,360	0.921
stringsearch	147,508	146,618	0.994	146,387	0.992	146,045	0.990
larger average	161,784	154,220	0.961	151,357	0.940	151,063	0.931
overall average	408,277	380,092	0.925	376,612	0.892	376,873	0.866

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. In contrast, WCET *code positioning* is performed after these required phases. Thus, there is no need to discard and reapply transformations after performing WCET *code positioning*.

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 7 and 8, WCET code positioning has a much greater impact on compilation time.

Table 7: Time Ratio After Superblock Formation and WC Path Duplication

Program	Superblock Formation	Path Duplication	Code Positioning
bubblesort	1.40	2.40	5.47
findmax	1.29	2.14	5.43
keysearch	1.17	2.08	5.83
summidall	1.43	2.57	6.71
summinmax	1.33	2.33	5.22
sumnegpos	1.43	2.29	6.71
sumoddeven	1.63	2.38	4.50
sumposclr	1.27	2.18	6.09
sym	1.30	2.50	5.90
unweight	1.38	2.13	5.88
small average	1.36	2.30	5.77
bitcnt	1.56	3.44	7.78
diskrep	1.75	3.38	17.44
fft	1.36	3.86	15.24
fire	1.67	4.00	15.33
sha	2.15	12.33	39.96
stringsearch	2.03	5.41	15.34
larger average	1.75	5.40	18.51
overall average	1.51	3.46	10.55

Table 8: Time Ratio After Loop Unrolling and Superblock Formation

Program	Loop Unrolling	Superblock Formation	Code Positioning
bubblesort	1.00	2.20	5.33
findmax	1.14	3.00	5.57
keysearch	1.08	1.75	3.75
summidall	1.29	2.57	9.29
summinmax	1.56	4.67	11.89
sumnegpos	1.14	5.57	20.00
sumoddeven	1.88	4.88	10.25
sumposclr	4.82	5.73	15.55
sym	1.10	1.90	4.20
unweight	1.25	2.88	6.50
small average	1.63	3.51	9.23
bitcnt	1.06	2.61	5.22
diskrep	2.38	11.75	29.69
fft	1.07	2.00	5.60
fire	1.22	4.00	19.78
sha	1.06	4.08	14.42
stringsearch	1.25	4.19	14.19
larger average	1.34	4.77	14.81
overall average	1.52	3.99	11.33

9 Future Work

In this paper, we performed WC path optimizations based on the WC path information from the timing analysis. There are many areas of future work that can be investigated to enhance these WCET path optimizations.

The compiler and the timing analyzer are currently separate processes and they exchange data via files. If we could merge the compiler and the timing analyzer into one process, then it would speed up the compilation since most of the compilation time is spent on timing analysis.

Currently, the compiler performs WC path optimizations on the innermost loops since they are considered to have the best impact on WCET for the smallest code size duplication. The timing analyzer has the ability to identify the critical code portion of a program. The compiler could concentrate on the code portion that has the most impact on WCET, instead of always attempting optimizations on the innermost loop since some inner loops may be in paths that will not affect the WCET.

Path optimizations reduce the WCET at the expense of an increase in code size. Currently, the compiler discards the code duplication if there is no improvement on WCET. However, it sometimes commits a large code size increase for small reductions on WCET. We could enhance the compiler to automatically weight the code size increase and WCET reduction to obtain the best choice. Alternatively, a user could specify the ratio of the code size increase to the WCET decrease that he/she is willing to accept.

10 Conclusions

In this paper we have described how the WCET of a program can be reduced by optimizing the WC paths. Compiler transformations that 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 the timing analyzer to detect the WCET paths through a function. We show 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 loop unrolling for an odd number of iterations without overhead, to improve WCET while minimizing code growth.

Code sinking and other conventional optimizations are applied on the WC path to further reduce its execution time. WCET code positioning is also performed at the final stage to further reduce the WCET. Since path optimizations may increase the code size, it was critical to obtain WCET

feedback from the timing analyzer to ensure that each code size increasing transformation improves the WCET before allowing it to be committed. Finally, we were able to show that these WC path optimizations improve ACET as well as WCET.

During the course of this research, we found 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 require only slightly more execution time 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.

11 Acknowledgements

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, CCR-0312531, and CCR-0312695.

References

Arnold, R., Mueller, F., and Whalley, D. 1994. Bounding worst-case instruction cache performance. In *Proceedings of the Fifteenth IEEE Real-time Systems Symposium*. IEEE Computer Society Press, San Juan, Puerto Rico, 172–181.

Benitez, M. 1994. Retargetable register allocation. Ph.D. thesis, University of Virginia, Charlottesville, VA.

Benitez, M. E. and Davidson, J. W. 1988. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*. ACM Press, Atlanta, Georgia, 329–338.

Benitez, M. E. and Davidson, J. W. 1994. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 International Conference on Programming Languages and Architectures*. 105–124.

Eyre, J. and Bier, J. 1998. Dsp processors hit the mainsteam. *IEEE Computer 31,* 8 (Aug.), 51–59.

Fisher, J. 1981. Trace scheduling: A technique for global microcode commpaction. *IEEE Transactions on Computers 30,* 7 (July), 478–490.

Gupta, R., Berson, D., and Fang, J. 1997. Path profile guided partial dead code elimination using prediction. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. ACM Press, San Francisco, California, 102–115.

Harmon, M., Baker, T., and Whalley, D. 1994. A retargetable technique for prediction execution time of code segments. *Real-Time Systems*, 159–182.

Healy, C., Arnold, R., Mueller, F., Whalley, D., and Harmon, M. 1999. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers 48,* 1 (Jan.), 53–70.

Healy, C., Sjodin, M., Rustagi, V., Whalley, D., and van Engelen, R. 2000. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems 18,* 2 (May), 121–148.

Healy, C. and Whalley, D. 1999. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, Vancouver, Canada, 79–99.

Healy, C. and Whalley, D. 2002. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering 28,* 8 (August), 763–781.

Healy, C., Whalley, D., and Harmon, M. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the Sixteenth IEEE Real-time Systems Symposium*. IEEE Computer Society Press, Pisa, Italy, 288–297.

Healy, C., Whalley, D., and van Engelen, R. 1999. A general approach for tight timing predictions of non-rectangular loops. In *WIP Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, Vancouver, CA, 11–14.

Hong, S. and Gerber, R. 1993. Compiling real-time programs into schedulable code. In *Proceedings of the SIGPLAN'93*. ACM Press, Albuquerque, New Mexico, 166–176.

Hwu, W., Mahlke, S., Chen, W., Change, P., Warter, N., Ouellette, R. B. R., Hank, R., Kiyohara, T., Haab, G., Holm, J., and Lavery, D. 1993. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing 7,* 1 (Mar.), 229–248.

Ko, L., Al-Yaqoubi, N., Healy, C., Ratliff, E., Arnold, R., Whalley, D., and Harmon, M. 1999. Timing constraint specification and analysis. *Software Practice & Experience 29,* 1 (Jan.), 77–98.

Ko, L., Healy, C., Ratliff, E., Arnold, R., Whalley, D., and Harmon, M. 1996. Supporting the specification and analysis of timing constraints. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*. IEEE Computer Society Press, Boston, Massachusetts, 170–178.

Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. 2003. Finding effective optimization phase sequences. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, San Diego, California, 12–23.

Lee, S., Lee, J., Park, C., and Min, S. 2004. A flexible tradeoff between code size and wcet using a dual instruction set processor. In *International Workshop on Software and Compilers for Embedded Systems*. Springer, Amsterdam, Netherlands, 244–258.

Mueller, F. 1997. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-time Systems*. ACM Press, Las Vegas, Nevada, 29–36.

Mueller, F. 2000. Timing analysis for instruction caches. *Real-Time Systems 18,* 2 (May), 209–239.

Mueller, F. and Whalley, D. 1992. Avoiding unconditional jumps by code replication. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation*. ACM Press, San Francisco, California, 322–330.

Mueller, F. and Whalley, D. 1995. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Languages Design and Implementation*. ACM Press, La Jolla, California, 55–56.

Star Core, I. 2001a. Sc100 simulator reference manual.

Star Core, I. 2001b. Sc110 dsp core reference manual.

T. Marlowe, S. M. 1992. Safe optimization for hard real-time programming. In *Special Session on Real-Time Programming, Second International Conference on Systems Integration*. 438–446.

White, R., Mueller, F., Healy, C., Whalley, D., and Harmon, M. 1999. Timing analysis for data caches and wrap-around-fill caches. *Real-Time Systems 17,* 1 (Nov.), 209–233.

White, R. T., Mueller, F., Healy, C., Whalley, D., and Harmon, M. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*. IEEE Computer Society Press, Montreal, Canada, 192–202.

Zhao, W., Cai, B., Whalley, D., Bailey, M., van Engelen, R., Yuan, X., Hiser, J., Davidson, J., Gallivan, K., and Jones, D. 2002. Vista: A system for interactive code improvement,. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, Berlin, Germany, 155–164.

Zhao, W., Kulkarni, P., Whalley, D., Healy, C., Mueller, F., and Uh, G. 2004. Tuning the wcet of embedded applications. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Toronto, Canada, 472–480.

Zhao, W., Whalley, D., Healy, C., and Mueller, F. 2004. Wcet code positioning. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Lisbon, Portugal, 81–91.

Zhao, W., Whalley, D., Healy, C., and Mueller, F. 2005. Improving wcet by applying a wc code positioning optimization. *ACM Transactions on Architecture and Code Optimization*, 335–365.