THE FLORIDA STATE UNIVERSITY

COLLEGE OF ART & SCIENCES

REDUCING THE WCET OF APPLICATIONS ON LOW END EMBEDDED SYSTEMS

By

WANKANG ZHAO

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2005

The members of the Committee approve the dissertation of Wankang Zhao defended on July 11, 2005.

David Whalley
Professor Directing Dissertation

Anuj Srivastava
Outside Committee Member

Theodore P. Baker
Committee Member

Robert A. van Engelen
Committee Member

Kyle Gallivan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGMENTS

I am deeply indebted in gratitude to my major professor, Dr. David Whalley, for his guidance, support, patience, and promptness during my research. He was always available to discuss new problems and exchange ideas. I cannot complete this dissertation without his excellent teaching and mentoring. I thank my committee members Dr. Baker, Dr. van Engelen, Dr. Gaillivan, and Dr. Srivastava for reviewing this dissertation and subsequent valuable suggestions. I also thank Prasad Kulkarni, William Kreahling, Stephen Hines, Richard Whaley, Gang-Ryung Uh and Frank Meuller for their assistance. The timing analyzer was developed by Chris Healy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Applications in embedded systems often need to meet specified timing constraints. It is advantageous to not only calculate the Worst-Case Execution Time (WCET) of an application, but to also perform transformations that attempt to reduce the WCET, since an application with a lower WCET will be less likely to violate its timing constraints.

A compiler has been integrated with a timing analyzer to obtain the WCET of a program on demand during compilation. This environment is used to investigate three different types of compiler optimization techniques to reduce WCET. First, an interactive compilation system has been developed that allows a user to interact with a compiler and get feedback regarding the WCET. In addition, a genetic algorithm is used to automatically search for an effective optimization phase sequence to reduce the WCET. Second, a WCET code positioning optimization has been investigated that uses worst-case path information to reorder basic blocks so that the branch penalties can be reduced in the worst-case path. Third, WCET path optimizations, similar to frequent path optimizations, are used to reduce the WCET.

There are several contributions to this work. To the best of our knowledge, this is the first compiler that interacts with a timing analyzer to use WCET predictions during the compilation of applications. The dissertation demonstrates that a genetic algorithm search can find an optimization sequence that simultaneously improves both WCET and code size. New compiler optimizations have been developed that use WC path information from a timing analyzer. The results show that the WCET code positioning algorithm typically finds the optimal layout of the basic blocks with the minimal WCET. It is also shown that frequent path optimizations can be applied on WC paths using worst-case path information from a timing analyzer to reduce WCET. These new compiler optimizations described in this dissertation not only significantly reduce WCET, but also are completely automatic.

CHAPTER 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) 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) of an application must be calculated to determine if its timing constraint will always be met. Simply measuring the execution time is not safe since it is difficult to determine input data that will cause the executable to produce the WCET. Accurate and safe WCET predictions can only be obtained by a tool that statically and correctly 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 predict the WCET accurately, 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.

WCET constraints can impact power consumption as well. Reducing the WCET of a task may allow an embedded system developer to use a lower clock rate to save additional power, which is valuable for mobile applications. The energy consumption E by an application is proportional to V^2ft, where V is the supply voltage, f is the clock frequency, and t is the time taken to run the application [1]. If the WCET is reduced by 10%, the clock rate can be lowered by 10% while still meeting the deadline. Setting a lower clock rate can also lower the supply voltage V, which reduces the power consumption quadratically. It will be shown in this dissertation that the WCET compiler optimizations that we have applied will typically reduce ACET as well. Even when the ACET increases, if this increase does not outweigh the benefit of the reduction in the supply voltage, then the WCET optimization will still reduce the energy consumption E.

Dynamic voltage scaling (DVS) is a technique to reduce energy consumption by dynamically adjusting the clock rate as low as possible while still making an application meet its timing constraint. However, the number of voltage levels for a processor is limited and the granularity of changing the voltage has overhead. In addition, many processors do not support DVS.

One common embedded system is a *digital signal processor* (DSP). Kernels for DSP applications have been historically written and optimized by hand in assembly code to ensure high performance [2]. However, assembly code is less portable, 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.

The execution time of an embedded application typically varies depending on the input data. The goal of most compiler optimizations is to improve the *average case execution time* (ACET), which is the typical execution time for a program, and/or code size. However, in this research, compiler optimizations are used to improve the WCET of an application, which is the maximum execution time of the application.

An interactive compilation system called VISTA (VPO Interactive System for Tuning Applications) [3, 4] is used to perform the experiments. VISTA allows users to develop embedded system applications in a high level language and still be able to tune the WCET of an application. VISTA has been integrated with a WC timing analyzer to automatically obtain WCET feedback information, which includes the WC paths and their WCETs. Figure 1.1 shows how the compiler obtains the WCET information after performing a sequence of optimizations. The compiler sends information about the control flow and the current instructions that have been generated to the timing analyzer. Predictions regarding the WCET and WC path information are sent back to the compiler from the timing analyzer to guide the compiler optimizations.

WCET compiler optimization has some advantages over ACET compiler optimization. First, some ACET compiler optimizations need profiling to detect the frequently executed

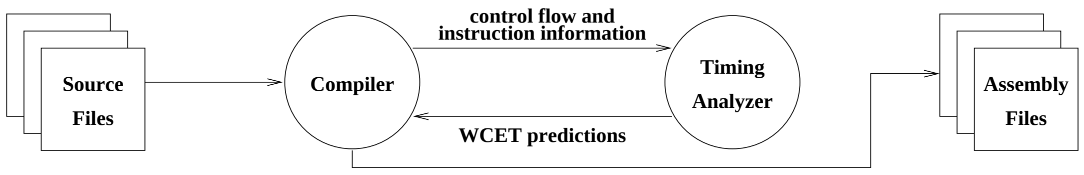


Figure 1.1. The Compiler Interacts with the Timing Analyzer to Obtain the WCET

paths to perform optimization. Profiling requires input data from the user that is supposed to represent average usage of the program. In contrast, WCET predictions do not rely on profiling and are totally automatic. Second, typically WCET compilation is faster than ACET compilation since performing timing analysis is faster than executing code. Furthermore, profiling on many embedded systems requires simulation, which can significantly slow down the compilation.

In this research, three different types of compiler optimization techniques have been developed to reduce the WCET of an application. First, a genetic algorithm is used to search for an effective optimization phase sequence that best reduces the WCET of the application. This method can reduce WCET and code size simultaneously when the fitness value for the genetic algorithm addresses both WCET and code size [5]. Second, a WCET code positioning optimization attempts to reduce WCET by reordering the basic blocks [6]. The number of unconditional jumps and taken branches that occur along the worst-case path is minimized by making basic blocks along the worst-case path contiguous in memory. Third, WCET path optimizations are used to reduce the WCET by code duplication. Traditional path-based compiler optimizations are used to reduce the ACET by optimizing frequently executed paths. Since these frequently executed paths may not be the *worst-case* (WC) paths, similar path optimizations on WC paths can be applied to reduce the WCET of a task [7]. The first method is a black box solution, in which the WC path information is not needed, while the last two methods are white box solutions, in which the WC path information is used to guide the optimizations.

The remainder of this dissertation has the following organization. Chapter 2 contains the prior work to this research, including WCET prediction and other WCET reduction techniques. The underlying architecture used for this research is the StarCore SC100.

Therefore, Chapters 3, 4 and 5 introduce the StarCore SC100 processor and how to retarget this machine to both the VPO (*very portable optimizer*) compiler and the timing analyzer used at FSU, respectively. The experimental environment in which this research was accomplished is presented in Chapter 6 and the benchmarks used are described in Chapter 7. The three compiler optimization techniques to reduce the WCET for embedded applications are discussed in the next three chapters. Chapter 8 presents applying a genetic algorithm to search for an effective optimization sequence that best reduces the WCET. Chapter 9 presents the code positioning algorithm to reduce the WCET for the optimal layout of the code in memory. Chapter 10 describes how to reduce WCET by adapting and applying path optimizations designed for frequent paths to WC paths in an application. Future work in this research area is outlined in Chapter 11 and the conclusions are given in Chapter 12.

CHAPTER 2

RELATED WORK

Research on reducing the WCET for embedded systems using compiler optimizations is a new area of research. However, this research is built upon prior work, such as WCET prediction techniques and compiler optimizations. This chapter summarizes the prior research on WCET prediction and reduction techniques. Prior research directly related to the three new WCET reduction techniques are presented in separate prior work sections in the chapters describing these techniques.

2.1 Prior Work on WCET Predictions

There has been much work on WCET predictions. Each of the general techniques for predicting WCET is discussed in the following subsections. Note that there are several other timing analysis techniques that are variations on the ones described in this section. Each of the general techniques has advantages and disadvantages. The reasons why the WCET prediction technique used in our timing analyzer is appropriate for performing WCET compiler optimizations are also discussed.

2.1.1 Timing Schema

In the 1980s, Shaw and Park started using timing schema to predict the maximum execution time of real-time programs [8, 9]. A WCET prediction tool has been developed later based on a timing schema, which is associated with each source-level program language construct [10]. In order to consider the effect of pipelining and caches on the execution time, the timing schema for each language construct contains WC pipeline and cache information. The timing bound for the whole program can be obtained by *concatenation* and *union* of the timing schema based on the source-level language constructs.

One limitation of this approach is that the prediction is associated with the source code instead of only the assembly. Compiler optimizations that change the control flow would invalidate the one-to-one correspondence between the source code and machine code. This limitation prevents the analysis of optimized code. Thus, the WCET from the timing schema cannot be easily used to guide the compiler optimizations to reduce the WCET.

2.1.2 Path Analysis

Another WCET prediction tool has been developed that is based on path analysis [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. The compiler provides the instruction and the control flow information. The path analysis is performed by the timing analyzer at the machine code level to calculate the execution time for each path in order to find the WC path. The execution time of a program is calculated by following a timing tree, where each node is a loop or a function.

This timing analyzer examines all the paths at each loop level to find the WC path. The analysis time is proportional to the number of paths at each loop level [17]. Since the path analysis technique to predict the WCET is performed at the machine instruction level, it is accurate and provides WCET path information that is useful for applying WCET compiler optimizations. Path analysis is used in this dissertation to predict the WCET of embedded applications. More details of this method are presented in Chapter 5.

2.1.3 Integer Linear Programming

The Integer Linear Programming (ILP) timing analysis method converts the problem of estimating WCET into a set of ILP constraints [25] [26]. After the ILP formulation, existing ILP solvers are used to determine the maximum total execution time. These ILP solvers can potentially determine a tight bound on a program's execution time.

The ILP approach is appealing since it is elegant and simple. However, it is common to have a large number of variables in the ILP constraints for even relatively small programs. It can be time-consuming to calculate the WCET using existing ILP solvers to deal with such a large number of variables. Furthermore, even if the WCET is obtained, only a single value representing the WCET is produced. This WCET prediction does not provide detailed information that can be used by compiler optimizations to improve the WCET.

2.1.4 Symbolic Execution

The symbolic execution method extends the instruction level architecture simulation technology with the capability of handling unknown input data values. Since the WC input data is unknown, it will appear that this approach would have to simulate all paths throughout the program to find the WCET. However, by using a path-merging strategy, this simulation technique can reduce the number of paths to be simulated [27]. Many paths are not simulated when the method finds that a branch must have a specific result. Symbolic execution can provide very accurate WCET prediction since it implicitly handles most functional constraints. But it can be very inefficient since it must simulate all loop iterations. Thus, the analysis time is proportional to the WCET of the program being analyzed. Such slow analysis would significantly increase compilation time if used in a compiler.

2.2 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 power 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 portions in a task [28]. They proposed that some conventional 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 their 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 [29]. 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 of these 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 generate code on a dual instruction set processor for the ARM and the Thumb [30]. ARM code is generated for a selected subset of basic blocks that can impact the WCET. Thumb code is generated for the remaining blocks. In this way, they can reduce the WCET while minimizing code size. They are using WCET information to choose the instruction set to select when generating code. In contrast, compiler optimizations are used to improve the WCET on a single instruction set processor.

CHAPTER 3

THE STARCORE SC100 ARCHITECTURE

The StarCore SC100, a Digital Signal Processor (DSP), is the architecture that is used in this research. The StarCore SC100 is a collaborative effort between Motorola and Agere and have been used in embedded system applications, such as DSL modems, wireless handsets, IP telephony, motor control, and consumer electronics. This low-to-middle performance DSP has the properties of compact code density, low power consumption and low cost [31]. There are no caches and no operating system in a SC100 system, which facilitates accurate WCET predictions.

In the StarCore SC100, there are three main functional units: the Program Sequencer Unit (PSEQ), the Data Arithmetic and Logic Unit (DALU), and the Address Generation Unit (AGU). The PSEQ performs instruction fetch, instruction dispatch, and exception processing. The DALU has one Arithmetic Logic Unit (ALU) to perform the arithmetic and logical operations, and the AGU has two Address Arithmetic Units (AAU) to perform address calculations. 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 machine has DSP addressing modes which are:

- register indirect: (Rn)
- post-increment: (Rn)+
- post-decrement: (Rn)-
- post-increment by offset: (Rn)+Ni

- index by offset: (Rn+N0)
- indexed: (Rn+Rm)
- displacement: (Rn+x)

The SC100 has a 5-stage pipeline. The five stages are:

- *Prefetch*: Generate the address for the fetch set and update the Program Counter (PC).
- *Fetch*: Read the fetch set from memory.
- *Dispatch*: Dispatch an instruction to the appropriate unit (AGU or DALU). Decode AGU instructions.
- *Address Generation*: Decode DALU instructions. Generate addresses for loads/stores. Generate target addresses for transfer of controls. Perform AGU arithmetic instructions.
- *Execution*: Perform DALU calculations. Update results.

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 and there are no branch prediction or target buffers that are used. 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. There are delayed change-of-flow instructions to allow filling-delay-slots optimizations. These delayed change-of-flow instructions require one less cycle of delay than the corresponding regular change-of-flow instructions. Transfers of control on this machine also incur an extra delay if the target is misaligned. The SC100

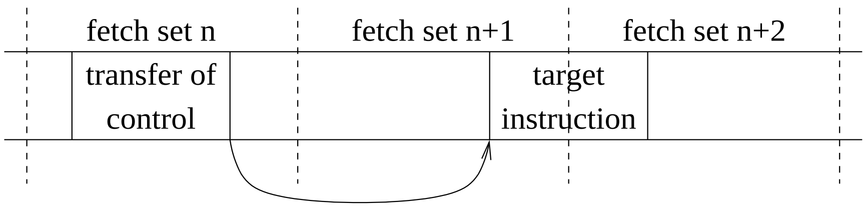


Figure 3.1. Example of a Misaligned Target Instruction

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 both in a different fetch set from the transfer of control and spans more than one fetch set, as shown in Figure 3.1. In this situation, the processor stalls for an additional cycle after the transfer of control [31].

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. Finally, the SC100 architecture does not provide hardware support for floating-point data types, nor does it provide divide functionality for integer types [31]. These operations are implemented by calling library functions.

CHAPTER 4

RETARGETING VPO TO THE STARCORE SC100

VISTA's optimization engine is based on VPO, the *very portable optimizer* [39, 40]. In order to determine the effectiveness of improving the WCET for applications on an embedded processor, the VPO compiler was ported to the StarCore SC100 processor. There are three components in the VPO compiler system: the front end, the code expander, and the backend, which are shown in Figure 4.1. VPO uses a very simple front end that translates the source code into intermediate code operations and performs no optimizations. A code expander is then used to translate these intermediate operations into a sequence of instructions represented by *register transfer lists* (RTLs). The code expander also does not perform any optimizations. The backend performs all optimizations on RTLs based on machine descriptions and generates the optimized assembly.

RTL is a low-level, machine- and language-independent representation that encodes machine-specific instructions. The comprehensive use of RTLs in VPO has several important consequences. Because there is a single representation, VPO offers the possibility of applying analyzes and optimizations repeatedly and in an arbitrary order. Therefore, it provides opportunities to tune the performance by reordering the optimization phases. In addition, the use of RTLs allows VPO to be largely machine-independent, yet efficiently handle machine-specific aspects such as register allocation, instruction scheduling, memory latencies, multiple condition code registers, etc. VPO, in effect, improves object code. Machine-specific optimization is important because it is a viable approach for realizing high-level language compilers that produce code that effectively balances target-specific constraints such as code density, power consumption, and execution speed.

Since the general form of RTLs is machine-independent, it is relatively easy to retarget VPO to a new machine, which is the reason why the VPO compiler is called the Very Portable

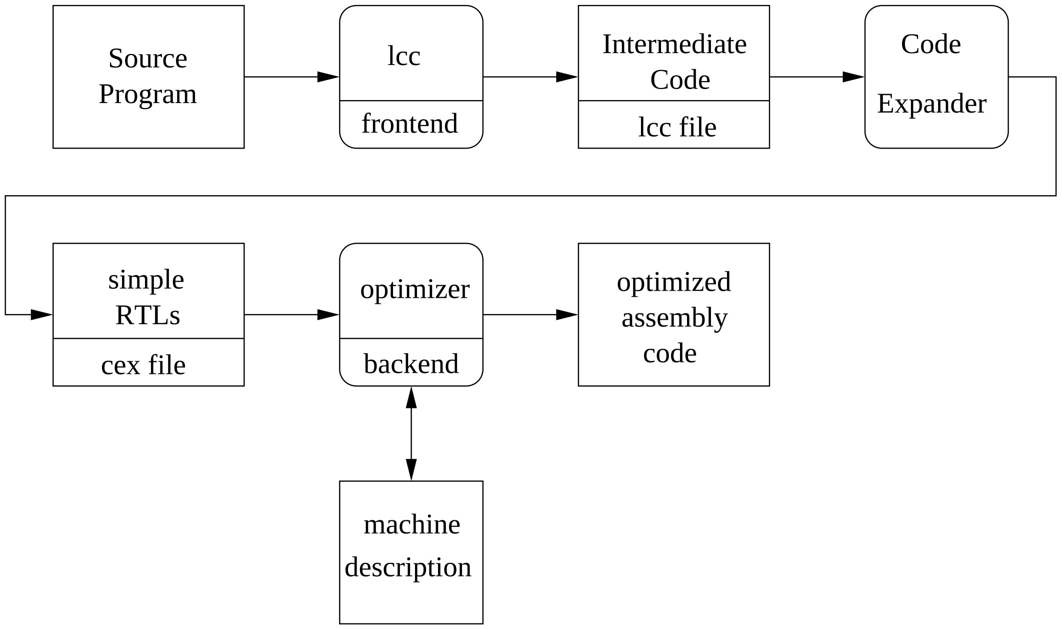


Figure 4.1. Overview of the VPO Compiler

Optimizer. Retargetability is key for embedded microprocessors where chip manufacturers provide many different variants of the same base architecture and some chips are custom designed for a particular application.

VPO often can be easily extended to handle new architectural features as they appear. Extensibility is also important for embedded chips where cost, performance, and power consumption considerations often mandate development of specialized features centered around a core architecture. The final property of VPO is that its analysis phases (e.g. data-flow analysis, control-flow analysis, etc.) were designed so that information is easily extracted and updated. This property makes writing new optimizations easier since data-flow and control-flow information can be collected by the analyzers.

Although all the three parts of the VPO compiler (shown in Figure 4.1) had to be modified to retarget the compiler for the SC100 processor, most of the retargeting effort was on the code expander and the backend. *lcc* is the compiler's front end [32], which translates C preprocessed source code into an *lcc* file containing machine-independent operations. The *lcc* file is fed into the code expander to produce the *cex* file, which contains the instructions represented by RTLs. The backend (optimizer) takes the *cex* file, performs optimizations, and generates the assembly file.

The procedure/function calling convention on this machine has a special feature. In most of the cases, the first two arguments are passed in registers, while the rest of the arguments are passed on a stack. However, functions with a variable number of arguments (varidatic functions) pass the last fixed argument and all subsequent variable arguments (varidatic arguments) on the stack. For example, if a varidatic function `printf()` has two arguments, then the first one is a fixed argument and the second one is a varidatic argument. Based on this rule, both arguments have to be passed on the stack. For another example, if a varidatic function `scanf()` has three arguments, then the first two arguments are fixed arguments and the third one is varidatic. The first argument has to be passed in a register and the remaining two arguments are passed on the stack. The *lcc* frontend had to be modified to indicate in the *lcc* file whether a function is a varidatic function and which arguments are fixed or varidatic.

The code expander should generate legal RTLs for the target machine. Since each machine has a different instruction set, the code expander has to be modified for each new architecture. The SC100 uses different types of registers to store data and address values, while most machines use the same type of registers for both. In addition, an SC100 conditional transfer of control consists of two instructions: a comparison instruction and a branch instruction. Different types of comparisons are encoded in the comparison instruction. Whether the second instruction jumps to the new target depends on the result (true/false) of the previous comparison instruction. The SC100 also has different mechanisms to handle multiply and division operations, parameter passing, etc. Furthermore, the data section in the assembly is generated in the code expander. The format of this section had to be made compatible with the SC100 data section specifications.

The backend of VPO consists of a machine-dependent part and a machine-independent part. In the machine-dependent part, the following tasks had to be performed to retarget it to the SC100.

- Change the machine description.
- Modify the calling convention for the SC100.

- Modify machine-dependent subroutines.

- Translate RTLs into SC100 assembly syntax.

VPO uses a syntax-directed translation tool called YACC to translate RTLs into assembly [41]. All legal RTLs for a machine are written as rules and are collectively called the machine description. Each YACC rule may be associated with an action to emit assembly code. Compiler optimizations, such as instruction selection, also need these rules to determine whether an RTL is legal or illegal for a particular machine. Even if the syntax of an RTL is legal, that RTL may not be a legal instruction. For example, *ADD #u5, Dn* is a SC100 instruction, where #u5 represents a 5-bit unsigned value and *Dn* represents a data register. A legal RTL for this instruction is *d[0]=d[0]+31;* while *d[0]=d[0]+32;* is illegal for the SC100 since the constant should be less than or equal to the largest 5-bit unsigned value for this instruction. Therefore, semantic checks for these instructions had to be added to the backend.

A calling convention defines how one procedure calls another. The calling convention varies for each machine. Information needed by a single activation of a procedure is managed by using a contiguous block of storage called an activation record or a frame. An SC100 activation record consists of local variables, temporaries, saved-registers, incoming/outgoing arguments and the return address. Figure 4.2 shows the stack frame layout for the SC100 [31]. At the entry point of a function, the stack pointer SP is increased by the size of the frame. At the exit point of the function, the SP is decreased back by the size of the frame. All the local variables are addressed by the SP minus an offset. The size of the frame is the summation of the space needed by incoming arguments, local variables, temporaries, saved registers, the return address, and outgoing arguments. If the function is not a leaf function, the outgoing arguments that cannot fit into registers are located in the space for outgoing arguments. If the function calls multiple functions, the size for outgoing arguments is determined by the callee function that needs the most space. The space for incoming arguments is used for storing the incoming arguments passed in registers. The incoming arguments passed on the stack are accessed by going to the caller function's frame. If non-scratch registers (d6, d7, r6 or r7) are used in this function, these registers have to be saved and restored at the entry and exit of the function.

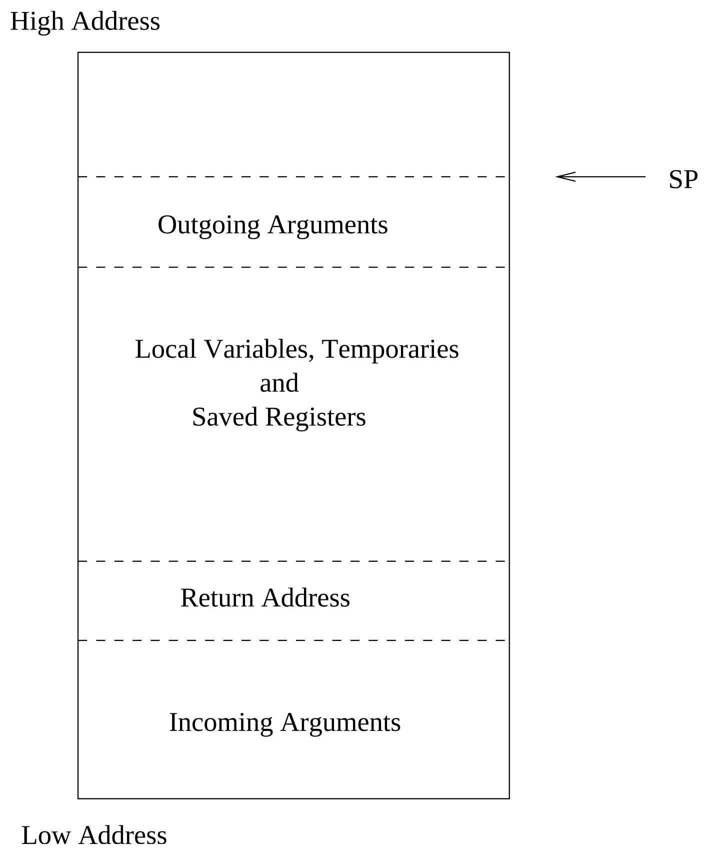


Figure 4.2. The Typical Layout of an Activate Record of SC100

Unlike most pipelined processors, the SC100 does not stall an instruction when the values on which it depends are not available. Since there are no pipeline interlocks, the last compiler optimization phase inserts 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 code size of each SC100 instruction may vary. The instruction type, the number of registered referenced, and the constant used can affect the size of an instruction. The VPO compiler was modified to enable it to statically calculate the code size of a program by inspecting these aspects of each instruction. This size information is used by the timing analyzer to detect the branch target misalignment delay. It is also used to measure the change on code size after applying optimizations in these experiments.

CHAPTER 5

RETARGETING THE TIMING ANALYZER TO THE SC100 PROCESSOR

The WCET of an application is obtained by a timing analyzer. This chapter summaries the timing analyzer [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] developed by FSU and how it was retargeted to the SC100 processor. Figure 5.1 depicts the organization of the framework that was used by the authors in the past to make WCET predictions. The compiler provides the instruction and the control flow information. The path analysis is performed by the timing analyzer at the machine code level to calculate the execution time for each path in order to find the WC path.

This path analysis approach for predicting WCET involves the following steps:

- architecture modeling (pipeline and cache)
- detecting the maximum number of iterations of each loop

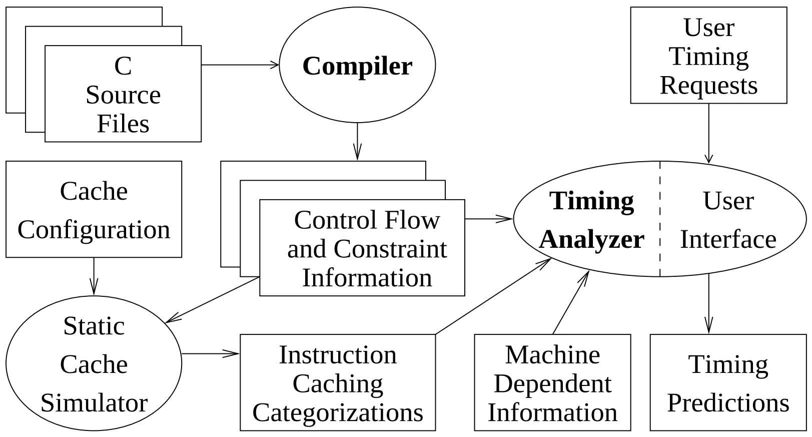


Figure 5.1. An Overview of the Existing Process to Obtain WCET Predictions

- 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 performing WC cache analysis [12]. A static cache simulator uses the control-flow information to give a caching categorization for each instruction and data memory reference in the program. The timing analyzer then integrates the cache analysis with pipeline analysis [13]. 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. Note that WCET prediction for many embedded machines is simpler since they often have a simple pipeline structure and a memory hierarchy consisting of ROM and RAM (no caches).

Besides addressing architectural features, the timing analyzer also automatically calculates control-flow constraints to tighten the WCET [18]. 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 [19]. Another type of constraint is branch constraints [24]. The timing analyzer uses these constraints to detect infeasible paths through the code and the frequency of how often a given path can be executed. 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 for a loop is calculated by repeatedly detecting the WCET path until a fixed point is reached where the caching behavior remains the same. The WCET of the whole program is calculated 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 are up to 2^{20} paths, which is not practical for path analysis 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. The timing tree is also updated to include each section as a direct descendant to the loop or function containing the section[17].

The architectural features of a machine, such as pipelining and caches, affect the WCET prediction of an application. The timing analyzer was retargeted to the SC100 processor, a DSP, to demonstrate that the WCET could be predicted for an embedded machine. It is difficult to produce cycle-accurate simulations for a general-purpose processor due to the complexity of its memory hierarchy and its interaction with an operating system that can cause execution times to vary. Fortunately, unlike most general-purpose processors, the SC100 does not have a memory hierarchy (no caches or virtual memory system). This simplifies the timing analysis for the SC100 since each instruction could be fetched in a single cycle if it is within one fetch set.

There are several modifications to support timing analysis of applications compiled for the SC100 processor. First, the machine-dependent information (see Figure 5.1) was modified to indicate how instructions proceed through the SC100 pipeline. Most SC100 instructions take one cycle to execute. However, some instructions require extra cycles in the pipeline. For instance, if an instruction uses an *indexed* or *displacement* memory addressing mode, it requires one additional cycle since the address has to be calculated from an arithmetic expression. Second, 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. Thus, the static cache simulation step shown in Figure 5.1 is now bypassed for the SC100. Third, the timing analyzer was 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 an transfer-of-control penalty, the timing analyzer calculates the number of clock cycles for the penalty, 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 detect when the target misalignment penalty will occur.

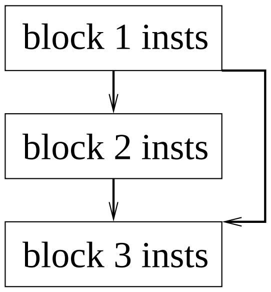


Figure 5.2. Example: Branch Penalty Affects the Worst-Case Path

In addition, we were able to obtain a simulator for the SC100 from StarCore [33]. 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 can simulate SC100 executables and report an estimated number of execution cycles for a program. This simulator is used to verify the accuracy of the WCET timing analyzer. This simulator, which can report the size of each instruction as well, is also used to verify the instruction sizes obtained from the compiler.

The transfer of control and branch target alignment penalties for SC100 can lead to nonintuitive WCET results. For instance, consider the flow graph in Figure 5.2. A superficial inspection would lead one to believe that the path $1 \rightarrow 2 \rightarrow 3$ is the WCET path through the graph. However, if the taken branch penalty in the path $1 \rightarrow 3$ outweighs the cost of executing the instructions in block 2, then $1 \rightarrow 3$ would be the WCET path. Simply measuring the execution time is not safe since it is very difficult to manually determine the WC paths and the input data that will cause the execution of these paths. This simple example illustrates the importance of using a timing analyzer to calculate the WCET.

CHAPTER 6

OVERALL FRAMEWORK OF THE RESEARCH

The research framework is built upon an interactive compilation system called VISTA [3, 4]. The development of VISTA was the topic of my master's thesis. VISTA was retargeted to the SC100 architecture and integrated it with the SC100 timing analyzer, which provides the WCET path information to the compiler to perform the WC compiler optimizations. Figure 6.1 shows an overview of the infrastructure used for this research. Note that Figure 5.1 only shows the timing analysis component in Figure 6.1. The SC100 compiler takes the source code and generates the assembly and the timing information. The timing information is fed into the timing analyzer to get the WC path information. The compiler performs the compiler optimizations to reduce WCET and the user can see transformations to the program graphically at the machine code level. The compiler automatically invokes the timing analyzer to update the WC path information, which is used to guide the next optimization. The graphical user interface in the viewer helps the user to see the progress on the improvement of WCET of an application, and it also was useful for debugging problems when developing the compiler optimization algorithms.

When the WCET is needed for a benchmark, VPO generates a timing information file for the benchmark and invokes the timing analyzer. The timing analyzer uses the timing information file as input to predict the WCET. Each instruction in the assembly has a counterpart in the timing information file where the instruction type and the registers set and used are presented. The timing analyzer needs this information to detect pipeline hazards and find the worst-case paths. The timing information file 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 file contains the control-flow information, such as the branches and loop, for the program. Many

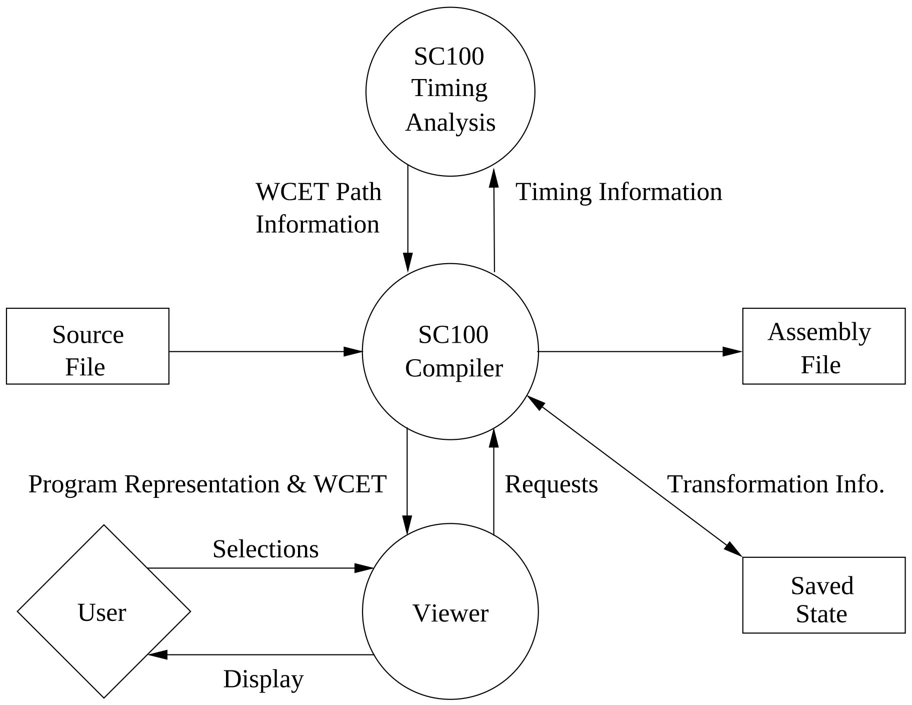


Figure 6.1. An Overview of the Research Framework

modifications to the SC100 VPO compiler were required to produce the timing information file for timing analysis.

The timing analyzer takes the timing information file and produces the WCET path information into another output file. VPO then reads the output file to get WCET information. WCET path optimizations need to be driven by the WCET path information obtained from the timing analyzer. The output file from the timing analyzer includes all the information about the timing tree. The information contains:

1. all the timing nodes (loops or functions)
2. parents/children relationship information for these nodes
3. function where each node is located
4. all paths in each node and their WCETs
5. all the basic blocks along each path

The 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 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 to guide the optimizations.

The compiler has the ability to reverse previously applied transformations. VPO compiler optimizes one function at a time. When the transformations need to be reversed, the current function and all its data structures are discarded and re-initialized. A fresh instance of the function is then read from the input file. The optimizations are re-applied up to a certain point. Therefore, the transformation information is saved to a file to prepare for the reverse. When the timing analyzer measures the WCET after one path optimization, transformations are reversed to the point before or after this path optimization is applied, depending whether or not the path optimization is retained.

CHAPTER 7

BENCHMARKS

Table 7.1 shows the benchmarks and applications used in these experiments for this dissertation. These programs include a subset of the *DSPStone* fixed-point kernel benchmarks and other benchmarks used in previous studies by various groups (FSU, SNU, Uppsala) working on WCET timing analysis. Many DSP applications have been historically optimized in assembly code by hand to ensure high performance [2]. In contrast, all of the results in this section are from code that was automatically generated by VISTA. The only *DSPStone* fixed-point kernel benchmarks not included are those that could not be automatically processed by the timing analyzer. In particular, the number of iterations for loops in some benchmarks could not be statically determined by the compiler. While the research framework allows a user to interactively supply this information, such programs are excluded to facilitate automating the experiments. Note that the *DSPStone* fixed-point kernel benchmarks are usually small and do not have conditional constructs, such as if statements. Other benchmarks shown in Table 7.1 were selected since they do have conditional constructs, which means they have more than one path and the WCET and ACET input data may not be the same.

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. If the input data for the original benchmark was from a file, then the benchmark was modified 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. If the benchmark has more than one path, the WC input data for the simulator has to be meticulously determined since the WC paths were often difficult to detect manually due to control-flow penalties.

Table 7.1. Benchmarks Used in the Experiments

Category		Program	Description
Small	DSPStone	convolution	performs a convolution filter
		complex_update	performs a single mac operation on complex values
		dot_product	computes the product of two vectors
		fir	performs a finite impulse response filter
		fir2dim	performs a finite impulse response filter on a 2D image
		iir_biquad_one_section	performs an infinite impulse response filter on one section
		iir_biquad_N_sections	performs an infinite impulse response filter on multiple sections
		lms	least mean square adaptive filter
		matrix	computes matrix product of two 10x10 matrices
		matrix_1x3	computes the matrix product of 3x3 and 3x1 matrices
		n_complex_updates	performs a mac operation on an array of complex values
		n_real_updates	performs a mac operation on an array of data
		real_update	performs a single mac operation
	Other	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

Table 7.2 shows the baseline of the experiments. The measurements are taken after all conventional optimizations have been applied. These benchmarks are classified into two categories: *small* and *larger* Benchmarks. The *small* benchmarks have either only one path (*DSPStone* benchmarks) or simple enough control-flow where the WC input data can likely be manually detected (*small* non-*DSPStone* benchmarks). The *larger* benchmarks have complicated control-flow where it is difficult to obtain the WC input data.

Table 7.2 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 the *small* non-*DSPStone* benchmarks. Although the code size of some of the *DSPStone* benchmarks is larger than 200 bytes, they are classified as *small* benchmarks since they have a single path.

The *observed ACET* is obtained from the simulator when random input data is used and the *observed WCET* is obtained from the SC100 simulator when WC input data is used. For *DSPStone* benchmarks, the *observed ACET* and the *observed WCET* are equal since there is only one path. For the *larger* benchmarks, the *observed WCET* is replaced with the *observed ACET* since it is very difficult to obtain the WC input data. The *observed ACET* and the *observed WCET* are different for most *small* non-*DSPStone* benchmarks in Table 7.2. The *observed ACET* and the *observed WCET* are very close for some of these benchmarks, while some are quiet different. The *observed ACET* of *bubblesort* is somewhat smaller than the *observed WCET* since with random data there is not an exchange for each comparison. Thus, the WC path is not taken each time. The *observed ACET* and the *observed WCET* are equal for *summinmax* since the two paths through *summinmax* have the same execution time. Therefore, the changes of paths driven by random input data do not affect the execution time. The *observed ACET* of *sym* is much smaller the *observed WCET* since the random input data makes the loops inside this benchmark break much earlier than with WC input data.

The *predicted WCET* is obtained from the timing analyzer. The *WCET Ratio* is the *predicted WCET* divided by the *observed WCET*. The *predicted WCET* should be larger than or equal to the *observed WCET* since the *predicted WCET* is the upper bound for the execution time and it should never be underestimated. The *predicted WCET* from the timing analyzer should be close to the execution time obtained from the simulator if the

WC input data is used in the simulation. However, the WC input data is too 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 the *small* programs, but much larger on average than the *observed cycles* for the *larger* benchmarks.

Table 7.2. The Baseline Code Size, Observed Cycles, and WCET

Category		Benchmarks	Code Size (bytes)	Lines of source	Observed ACET	Observed WCET	Predicted WCET	WCET Ratio
Small	DSPStone	convolution	87	71	635	635	642	1.011
		complex_update	169	80	152	152	157	1.033
		dot_product	80	79	118	118	127	1.076
		fir	149	85	1,082	1,082	1,087	1.005
		fir2dim	370	149	5,518	5,518	5,756	1.043
		iir_biquad_one_section	150	88	122	122	128	1.049
		iir_biquad_N_sections	219	119	1,054	1,054	1,070	1.015
		lms	237	160	1,502	1,502	1,513	1.007
		matrix	216	134	37,012	37,012	37,364	1.010
		matrix_1x3	89	77	262	262	279	1.065
		n_complex_updates	294	92	2,993	2,933	2,938	1.002
		n_real_updates	153	73	1,570	1,570	1,577	1.004
		real_update	68	68	79	79	85	1.076
	Other	bubblesort	125	93	5,086,184	7,365,289	7,616,299	1.034
		findmax	58	21	19,991	19,997	20,002	1.000
		keysearch	189	537	11,247	31,164	31,768	1.019
		summidall	56	23	19,511	19,513	19,520	1.000
		summinmax	62	47	23,011	23,011	23,017	1.000
		sumnegpos	45	20	18,032	20,010	20,015	1.000
		sumoddeven	78	51	14,783	22,025	23,032	1.046
		sumposclr	81	35	28,469	31,013	31,018	1.000
		sym	97	40	107	55,343	55,497	1.003
		unweight	79	23	340,577	350,507	350,814	1.001
small average			137	94	244,085	347,387	358,422	1.022
Larger		bitcnt	355	170	40,416	40,416	58,620	1.450
		diskrep	379	500	10,028	10,028	12,661	1.263
		fft	647	220	76,505	76,505	76,654	1.002
		fire	250	109	8,900	8,900	10,211	1.147
		sha	909	253	691,047	691,047	769,495	1.114
		stringsearch	343	237	148, 849	148,849	195,991	1.317
larger average			481	248	162,624	162,624	187,272	1.215
overall average			309	171	203,354	255,006	272,847	1.119

CHAPTER 8

TUNING THE WCET BY SEARCHING FOR EFFECTIVE OPTIMIZATION SEQUENCES

In this chapter, an approach for reducing the WCET of an application by using an interactive compilation system called VISTA [3, 4] to find a sequence of compiler optimization phases that best reduces WCET is presented [5]. VISTA allows users to develop embedded system applications in a high level language and still be able to tune the WCET of an application. One feature of VISTA is that it can automatically obtain WCET performance feedback information, which can be used by both the application developer and the compiler to make phase ordering decisions. In addition, this system allows a user to invoke a genetic algorithm that automatically searches for an effective optimization phase sequence for each function that best reduces the WCET. VISTA applies a sequence of optimization phases on the current function and measures the WCET using the timing analyzer. The transformations applied are then reversed. In this way, additional sequences of optimization phases can be evaluated. The best sequence of optimization phases encountered is then reapplied after the genetic algorithm completes.

The remainder of this chapter has the following organization. Prior research on visualization of the compilation process, interactive compilation systems, and searching for effective optimization sequences is first summarized. Our interactive compilation system supporting selecting an optimization sequence to reduce the WCET is then described. Finally, the challenges and the experimental results to perform the search are given.

8.1 Prior Work on Visualization of the Compilation Process and Interactive Compilation

There exist systems that are used for simple visualization of the compilation process. The *UW Illustrated Compiler* [35], also known as *icomp*, has been used in undergraduate compiler classes to illustrate the compilation process. The XVPODB system [36] [37] has been used to illustrate low-level code transformations in the VPO compiler system [38]. XVPODB has also been used when teaching compiler classes and to help ease the process of re-targeting the compiler to a new machine or diagnosing problems when developing new code transformations.

Other researchers have developed systems that provide interactive compilation support. These systems include the *pat* toolkit [42], the *parafrase-2* environment [43], the *e/sp* system [44], a visualization system developed at the University of Pittsburgh [45], and the *SUIF explorer* [46]. These systems provide support by illustrating the possible dependencies that may prevent parallelizing transformations. A user can inspect these dependencies and assist the compilation system by indicating if a dependency can be removed. In contrast, VISTA does not specifically deal with parallelizing transformations, but instead supports low-level transformations and user-specified changes, which are needed for tuning embedded applications in general.

A few low-level interactive compilation systems have also been developed. One system, which is coincidentally also called VISTA (*Visual Interface for Scheduling Transformations and Analysis*), allows a user to verify dependencies during instruction scheduling that may prevent the exploitation of instruction level parallelism in a processor [47]. Selective ordering of different optimization phases does not appear to be an option in their system. The system that most resembles the current work is called VSSC (*Visual Simple-SUIF Compiler*) [48]. It allows optimization phases to be selected at various points during the compilation process. It also allows optimizations to be reversed, but unlike the current compiler, only at the level of complete optimization phases as opposed to individual transformations within each phase. Our VISTA system [3, 4] supports user-specified changes and performance feedback information, which does not appear to be available in these systems. Furthermore, all of these other systems are not integrated with a WCET timing analyzer. They are instead designed to improve ACET.

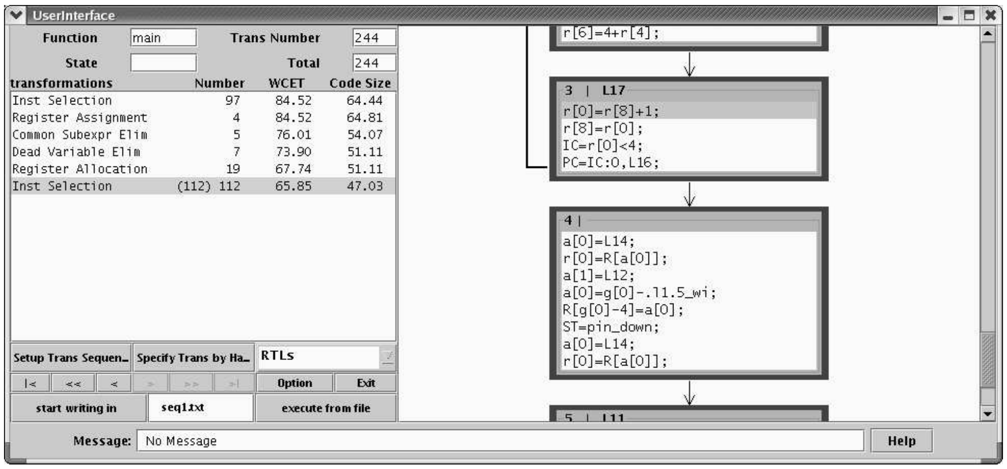
8.2 Prior Work on Searching for Effective Optimization Sequences

There has been prior work on aggressive compilation techniques to improve performance. *Superoptimizers* have been developed that use an exhaustive search to find a better instruction sequence. This instruction reordering search technique can be used for instruction selection [49] or to eliminate branches [50]. Selecting the best combination of optimizations by turning on or off optimization flags, as opposed to varying the order of optimizations, has also been investigated [51].

Iterative techniques using performance feedback information after each compilation have been applied to determine good optimization parameters (e.g., blocking sizes) for specific programs or library routines [52, 53, 54, 55]. Since the search space for the optimization parameters is too large, two steps were used to find an effective location in the search space. The first step used a coarse grid to find an area where the best performance appears to be located. A better location is found in the second step by using a fine grid around this area. SALTO is a framework for tuning embedded applications on low-level codes [56]. It enables the building of profiling, tracing and optimization tools. For example, the feedback on the execution time of the loop body is used as a guide for applying specific loop transformations, such as loop unrolling and loop fusion. These iterative algorithms searching for optimization parameters work well when the search space is smooth and points near each other in the space do not typically result in significant difference in performance. If the search space is not smooth, this search algorithm would not work well.

Genetic algorithms are search algorithms that are used to probe a non-contiguous (non-smooth) search space. A system using a genetic algorithm to better parallelize loop nests has been developed and evaluated [57]. A low-level compilation system developed at Rice University uses a genetic algorithm to reduce code size by finding efficient optimization phase sequences [58]. In the Rice experiments, they also studied other search techniques in an attempt to find a quality solution in less time [59].

VISTA also provides support for automatically using performance information to select an effective optimization sequence using a genetic algorithm. The performance criteria in previous experiments were static code size and the number of dynamic instructions executed [4]. In this work, VISTA has been integrated with a timing analyzer, and the WCET



Function: main   Trans Number: 244
State: Total: 244

transformations	Number	WCET	Code Size
Inst Selection	97	84.52	64.44
Register Assignment	4	84.52	64.81
Common Subexpr Elim	5	76.01	54.07
Dead Variable Elim	7	73.90	51.11
Register Allocation	19	67.74	51.11
Inst Selection	(112) 112	65.85	47.03

RTLs

seq1.txt

```
r[6]=4+r[4];

3 | L17
r[0]=r[8]+1;
r[8]=r[0];
IC=r[0]<4;
PC=IC:0,L16;

4 |
a[0]=L14;
r[0]=R[a[0]];
a[1]=L12;
a[0]=g[0]-.11.5_wi;
R[g[0]-4]=a[0];
ST=pin_down;
a[0]=L14;
r[0]=R[a[0]];

5 | L11
```

Message: No Message

Figure 8.1. WCET VISTA Snapshot

and/or code size are used as the performance criteria in the genetic algorithm [5]. Besides applying the genetic algorithm to the phase ordering problem, genetic algorithms have also been used in the context of timing analysis for empirically estimating the WCET, where mutations on the input resulted in different execution times (objective function) [60, 61, 62].

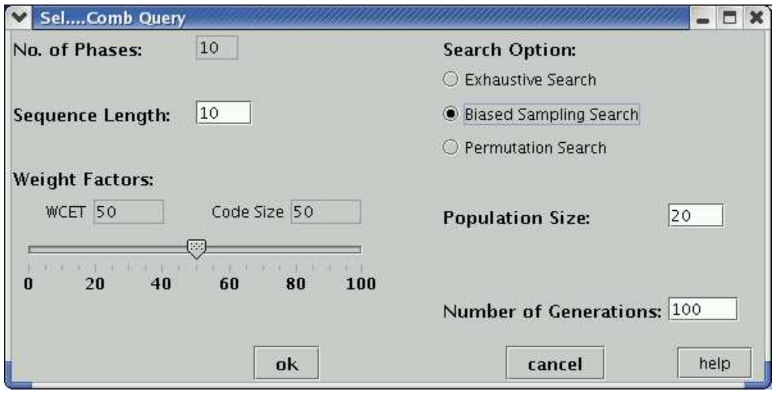
8.3 Our Approach

Figure 8.1 shows a snapshot of the viewer when tuning an application for the SC100. The right side of the window displays the state of the current function as a control-flow graph with RTLs representing instructions. The user also has the option to display the instructions in assembly. The left side shows the history of the different optimization phases that have been performed in the session. Note that not only is the number of transformations associated with each optimization phase depicted, but also WCET and code size improvements are shown in the figure. The WCET and code size improvements corresponding to each phase are the percentages of the WCET and code size after performing this phase compared with the WCET and code size before performing any compiler optimizations. Thus, a user can easily gauge the progress that has been made at tuning the current function. In addition, a genetic algorithm can be used to tune the WCET automatically.

A genetic algorithm is basically a search algorithm designed to mimic the process of natural selection and evolution in nature. Genetic algorithms have long been used to search for solutions in a space that is too large to be exhaustively evaluated. Traditional compilers usually apply a fixed sequence of optimization phases for all programs. However, the best sequence may be different for each function and the search space for finding an better sequence is too large. Therefore, genetic algorithms have been used in the past to search for effective optimization sequences to improve speed, space, or a combination of both factors for the programs being compiled [4, 58]. A chromosome, consisting of a sequence of genes, represents a sequence of optimization phases. Each gene in a chromosome represents an optimization phase. Each sequence of phases has to be evaluated based on a fitness value to determine whether it should be discarded or kept in the next generation. The crossover and mutation operations of the chromosomes in the candidate pool will generate new chromosomes.

In the current work, WCET and code size are used as a measurement of performance. To obtain the fitness value for a chromosome, the WCET and the code size of a program are measured after the optimization phases corresponding to the chromosome are applied. Changes to the program made by these optimization phases are then discarded and the program returns to the original state. Alternative sequences can be applied and evaluated in this way. The best sequence found by the algorithm would be applied again to obtain the version with the best fitness value.

Figure 8.2 shows the different options that VISTA provides the user to control the search. The experiments (described in Section 8.5) use the biased sampling search, which applies a genetic algorithm in an attempt to find the most effective sequence within a limited number of generations since in many cases the search space is too large to be exhaustively evaluated [63]. The sequence length is the total number of phases (genes) applied in each sequence (chromosome). A population is the set of solutions (sequences) that are under consideration. The number of generations indicates how many sets of populations are to be evaluated. The population size and the number of generations limit the total number of sequences evaluated. VISTA also allows the user to choose WCET and code size weight factors, where the relative improvement of each is used to determine the overall fitness.



No. of Phases: 10

Sequence Length: 10

Weight Factors:

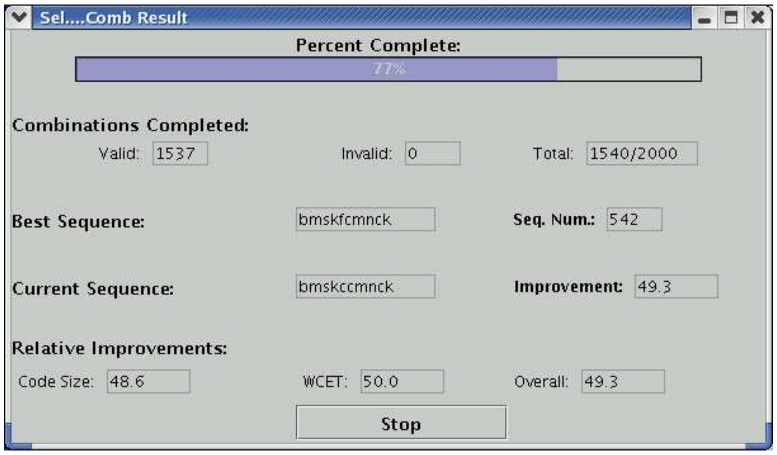
WCET 50 Code Size 50

Search Option:
( ) Exhaustive Search
(●) Biased Sampling Search
( ) Permutation Search

Population Size: 20

Number of Generations: 100

Figure 8.2. Selecting Options to Search for Sequences



Percent Complete: 77%

Combinations Completed:

Valid: 1537 Invalid: 0 Total: 1540/2000

Best Sequence: bmskfcmnck Seq. Num.: 542

Current Sequence: bmskccmnck Improvement: 49.3

Relative Improvements:

Code Size: 48.6 WCET: 50.0 Overall: 49.3

Figure 8.3. Window Showing the Search Status

Performing these searches can be time consuming since thousands of potential optimization sequences may need to be evaluated. Thus, VISTA provides a window showing the current status of the search. Figure 8.3 shows a snapshot of the status of the search. The percentage of sequences completed along with the best sequence and its effect on performance are displayed. The user can terminate the search at any point and accept the best sequence found so far.

8.4 Challenges to Obtaining the WCET in VISTA

There are challenges that have to be overcome to obtain the WCET in VISTA. Some required optimization phases have to be performed before measuring the WCET. The phase sequence selected by users may not contain these required phases. In addition, the number of iterations of all the loops have to be known in order to automatically measure the WCET. The compiler can detect the number of iterations automatically if the loop is bounded and five optimization phases, *register assignment, register allocation, instruction selection, loop strength reduction and fix entry exit*, have been done. Without performing these five phases, the timing analyzer cannot measure the WCET because the number of loop iterations is unknown to the compiler. Furthermore, the WCET of the whole program is calculated in a bottom up process by following a timing analysis tree, where each node in the tree is a loop or a function. Each function is treated as a loop with a single iteration and the WCET of a loop is not calculated until the WCETs of all its immediate child loops are known. However, VPO optimizes one function at a time in the order of the position of the function in the source file. As a result, when VPO starts to optimize one function, it may not have encountered all the functions called by this function and the WCETs for these functions are unknown. Without the WCETs for all the functions called by this function, the WCET of this function cannot be calculated.

For these reasons, three passes in VPO are used to reduce the WCET. The first pass performs the five required phases so that the number of iterations for each loop can be obtained. The compiler then reverses all the five phases. The second pass processes all the functions in the program, applying only the required phases (*register assignment*, *fix entry exit*, and *add no-ops*), generating the information file needed by the timing analyzer, and obtaining the initial WCETs before applying any optimizations for every function. These required phases performed by the second pass are also reversed. Finally, the third pass is used to tune the WCET. For each particular function, the number of loop iterations for the current function and the WCET for all the rest of the functions are available at this time. So after each optimization phase, the timing analyzer can measure the new WCET for the current function within the context of the initial WCETs of all other functions being known.

From a user's perspective, the first two passes can be considered as one pass. Therefore, there are only two passes visible to the users. The first pass obtains the loop information and the initial WCET. The second pass performs optimizations to reduce the WCET. In the first pass, both performing the required transformations and reversing these transformations are handled automatically. In the second pass, if the user does not select the required phases, then the compiler automatically performs these transformations before taking measurements.

The timing analyzer and the VPO compiler are separate processes. When the compiler needs to measure the WCET, it generates the timing information file. The compiler then invokes the timing analyzer, which takes the timing information file as input. The timing analyzer performs timing analysis and saves the results into another file. The compiler opens the output file of the timing analyzer and obtains the WCET.

When a user selects a sequence of phases to improve the WCET, the compiler will automatically invoke the timing analyzer after each phase to get the improvement in WCET for each step. Thus, the user can see the percentage improvement in WCET after each phase (see Figure 8.1). When the user selects the genetic algorithm to automatically search for the best sequence, the timing analyzer is only invoked after applying each optimization sequence, instead of after each phase. Thus, the genetic algorithm obtains the WCET for each sequence. Only when reapplying the best sequence found after completing the genetic algorithm is the timing analyzer invoked after each phase.

8.5 Experiments

This section describes the results of a set of experiments to illustrate the effectiveness of improving the WCET by using VISTA's biased sampling search, which uses a genetic algorithm to find efficient sequences of optimization phases. Tuning for ACET or WCET may result in similar code, particularly when there are few paths through a program. However, tuning for WCET can be performed faster since the timing analyzer is used to evaluate each sequence. Depending on the number of generations, thousands of optimization phase sequences may be required to search for the best sequence of optimization phases. The analysis time required for the timing analyzer is proportional to the number of unique paths at each loop and function level in the program. In contrast, obtaining a dynamic measure for the SC100 would require invoking the simulator, which is much slower. Tuning for ACET typically takes much longer since the execution time of the SC100 simulator is proportional to the number of instructions executed. The average time required to tune the WCET of most programs in the experiments was less than 15 minutes, as shown in Table 8.1, and this would have taken several hours if simulation had been used. Note that the baseline compilation time in the table is longer than the regular VPO compilation time since the compiler has to invoke the timing analyzer at the end of the compilation to obtain the baseline WCET.

Table 8.2 shows each of the candidate code-improving phases used in the experiments when tuning each function with the genetic algorithm. Each phase consists of one or more transformations. In addition, register assignment, which is a compulsory phase that assigns pseudo registers to hardware registers, has to be performed. VISTA implicitly performs register assignment before the first code-improving phase in a sequence that requires it. After applying the last code-improving phase in a sequence, another compulsory phase, fix entry/exit, which inserts instructions at the entry and exit of the function to manage the activation record on the run-time stack, is performed. Finally, additional code-improving phases after each sequence, such as instruction scheduling, is also performed. For the SC100 another compulsory phase is required to insert no-ops when pipeline hazards need to be addressed.

The genetic algorithm searches are accomplished in the following manner. The sequence (chromosome) length is set to be 1.25 times the number of phases that successfully applied one or more transformations by the batch compiler for the function. This number of phases is a reasonable limit and gives the compiler an opportunity to successfully apply more phases than what the batch compiler could accomplish. Note that this length is much less than the number of phases attempted during the batch compilation. The population size (fixed number of sequences or chromosomes) is set to twenty and each of these initial sequences is randomly initialized with candidate optimization phases. When searching for the best sequence for each function, 200 generations are performed. The sequences in the population are sorted by a fitness value based on the WCET produced by the timing analyzer and/or code size. At each generation (time step) the worst sequence is removed and three others from the lower (poorer performing) half of the population are chosen to be removed at random. Each of the removed sequences are replaced by randomly selecting a pair of

Table 8.1. The Compilation Time of the Benchmarks

Category		Benchmarks	Compilation Time		
			baseline(min)	GA(min)	Ratio
Small	DSPStone	convolution	0.28	1.60	5.65
		complex_update	0.43	2.32	5.35
		dot_product	0.28	1.42	5.00
		fir	0.33	2.13	6.40
		fir2dim	0.57	8.87	15.65
		iir_biquad_one_section	0.32	1.78	5.63
		iir_biquad_N_sections	0.67	12.98	19.48
		lms	0.42	3.75	9.00
		matrix	0.43	6.75	15.58
		matrix_1x3	0.23	2.10	9.00
		n_complex_updates	0.63	7.03	11.11
		n_real_updates	0.35	1.90	5.43
		real_update	0.27	1.37	5.13
	Other	bubblesort	0.55	6.60	12.00
		findmax	0.23	1.60	6.86
		keysearch	0.37	10.93	29.82
		summidall	0.27	2.62	9.81
		summinmax	0.25	4.13	16.53
		sumnegpos	0.25	1.55	6.20
		sumoddeven	0.28	5.82	20.53
		sumposclr	0.32	6.48	20.47
		sym	0.30	6.75	22.50
		unweight	0.25	2.03	8.13
small average			0.36	4.46	11.79
Larger		bitcnt	0.90	6.45	7.17
		diskrep	0.70	33.60	48.00
		fft	1.90	221.62	116.64
		fire	0.33	7.50	22.50
		sha	2.65	60.00	22.64
		stringsearch	1.18	18.12	15.31
larger average			1.28	57.88	38.71
overall average			0.55	15.51	17.36

Table 8.2. Candidate Optimization Phases in the Genetic Algorithm Experiments

Optimization Phase	Description
branch chaining	Replaces a branch or jump target with the target of the last jump in a jump chain.
common subexpr elim	Eliminates fully redundant calculations, which also includes constant and copy propagation.
remove unreachable code	Removes basic blocks that cannot be reached from the entry block of the function.
remove useless blocks	Removes empty blocks from the control-flow graph.
dead assignment elim	Removes assignments when the assigned value is never used.
block reordering	Removes a jump by reordering basic blocks when the target of the jump has only a single predecessor.
minimize loop jumps	Removes a jump associated with a loop by duplicating a portion of the loop.
register allocation	Replaces references to a variable within a specific live range with a register.
loop transformations	Performs loop-invariant code motion, recurrence elimination, loop strength reduction, and induction variable elimination on each loop ordered by loop nesting level. Each of these transformations can also be individually selected by the user.
merge basic blocks	Merges two consecutive basic blocks a and b when a is only followed by b and b is only preceded by a.
evaluation order determination	Reorders RTLs in an attempt to use fewer registers.
strength reduction	Replaces an expensive instruction with one or more cheaper ones.
reverse jumps	Eliminates an unconditional jump by reversing a conditional branch when it branches over the jump.
instruction selection	Combine instructions together and perform constant folding when the combined effect is a legal instruction.
remove useless jumps	Removes jumps and branches whose target is the following block.

the remaining sequences from the upper half of the population and performing a crossover (mating) operation to create a pair of new sequences. The crossover operation combines the lower half of one sequence with the upper half of the other sequence and vice versa to create two new sequences. Fifteen sequences are then changed (mutated) by considering each optimization phase (gene) in the sequence. Mutation of each optimization phase in the sequences occurs with a probability of 10% and 5% for the lower and upper halves of the population, respectively. When an optimization phase is mutated, it is randomly replaced with another phase. The four sequences subjected to crossover and the best performing sequence are not mutated. Finally, if identical sequences are found in the same population, then the redundant sequences are replaced with ones that are randomly generated. The characteristics of this genetic algorithm search are very similar to those used in past studies [4] [58], except the objective function now is minimizing the WCET.

Table 8.3 shows the WCET prediction results and the WCET reduction by using the genetic algorithm for the benchmarks. While each function was tuned separately, the WCET value shown in the table is for the whole program. The batch sequence results are those that are obtained from the sequence of applied phases by using VPO's default batch optimizer. The batch compiler iteratively applies optimization phases until there are no additional improvements. Thus, the batch compiler provides a much more aggressive baseline than a compiler that always uses a fixed length of phases. In Table 8.3, the *observed cycles* for *batch sequence* were obtained from running the compiled programs with WC input data through the SC100 simulator. The *WCET cycles* for the *batch sequence* are the WCET predictions obtained from the timing analyzer. The *ratio* column for the *batch sequence* in Table 8.3 is the result of the *WCET cycles* divided by the *observed cycles*. The *ratios* for the *best sequence from GA* results in Table 8.3 are similar, but the code being measured was after the best sequence found by the genetic algorithm being applied. The *GA to batch ratio* shows the ratio of WCET cycles after applying the genetic algorithm to the WCET cycles from the code produced by the batch sequence of optimization phases. Some applications, like *fft*, had significant improvements. The applications with larger functions tend to have more successfully applied phases, which can often lead to larger improvements when searching for an effective optimization sequence. The average number of generations to find the most effective sequence was 51 out of the 200 generations attempted. The larger the number of generations performed, the better the chance to find a more effective sequence of phases. For each function, 200 generations were performed since the search could be accomplished in a reasonable time for most benchmarks.

While there are some aberrations due the randomness of using a genetic algorithm, most of the benchmarks have improved WCETs. The WCET cycles on average decrease by 4.8% for the *Small* benchmarks and by 5.7% for the *Larger* benchmarks. For all the benchmarks, there is 5.2% reduction on average for WCET. This illustrates the benefit of using a genetic algorithm to search for effective optimization sequences to improve WCET.

Table 8.3. WCET Prediction Results and Reduction in WCET Using the Genetic Algorithm

Category		Program	Batch Sequence			Best Sequence from GA			GA to Batch Ratio
			Observed Cycles	WCET Cycles	Ratio	Observed Cycles	WCET Cycles	Ratio	
Small	DSPStone	convolution	635	642	1.011	603	610	1.012	**0.950**
		complex_update	152	157	1.033	150	154	1.027	**0.981**
		dot_product	118	127	1.076	107	115	1.075	**0.906**
		fir	1,082	1,087	1.005	986	991	1.005	**0.912**
		fir2dim	5,518	5,756	1.043	5,241	5,472	1.044	**0.951**
		iir_biquad_one_section	122	128	1.049	122	128	1.049	**1.000**
		iir_biquad_N_sections	1,054	1,070	1.015	881	897	1.018	**0.838**
		lms	1,502	1,513	1.007	1,406	1,417	1.008	**0.937**
		matrix	37,012	37,364	1.010	35,223	35,568	1.010	**0.952**
		matrix_1x3	262	279	1.065	252	267	1.060	**0.957**
		n_complex_updates	2,933	2,938	1.002	2,885	2,891	1.002	**0.984**
		n_real_updates	1,570	1,577	1.004	1,426	1,433	1.005	**0.909**
		real_update	79	85	1.076	73	77	1.055	**0.906**
	Other	bubblesort	7,365,289	7,616,299	1.034	7,119,775	7,339,847	1.031	**0.964**
		findmax	19,997	20,002	1.000	19,996	20,001	1.000	**1.000**
		keysearch	31,164	31,768	1.019	30,666	31,272	1.020	**0.984**
		summidall	19,513	19,520	1.000	18,516	18,521	1.000	**0.949**
		summinmax	23,011	23,017	1.000	23,009	23,015	1.000	**1.000**
		sumnegpos	20,010	20,015	1.000	19,013	19,018	1.000	**0.950**
		sumoddeven	22,025	23,032	1.046	22,028	22,052	1.001	**0.957**
		sumposclrneg	31,013	31,018	1.000	30,011	30,017	1.000	**0.968**
		sym	55,343	55,497	1.003	54,118	54,272	1.003	**0.978**
		unweight	350,507	350,814	1.001	340,308	340,714	1.001	**0.971**
small average			347,387	358,422	1.022	335,948	345,598	1.018	**0.952**
Larger		bitcnt	40,416	58,620	1.450	38,416	53,420	1.391	**0.911**
		diskrep	10,028	12,661	1.263	10,068	12,589	1.250	**0.994**
		fft	76,505	76,664	1.002	60,858	61,311	1.007	**0.800**
		fire	8,900	10,211	1.147	8,900	10,211	1.147	**1.000**
		sha	691,047	769,495	1.114	675,049	751,205	1.113	**0.976**
		stringsearch	148,849	195,991	1.317	147,154	191,597	1.302	**0.978**
larger average			162,624	187,272	1.215	156,741	180,056	1.202	**0.943**
average			255,006	272,847	1.119	246,344	262,827	1.110	**0.948**

Table 8.4. Effect on ACET When Optimizing for WCET

Category		Program	Batch Sequence ACET Cycles	Best Sequence from GA ACET Cycles	GA to Batch Ratio
Small	DSPstone	convolution	635	603	**0.950**
		complex_update	152	150	**0.987**
		dot_product	118	107	**0.907**
		fir	1,082	986	**0.911**
		fir2dim	5,518	5,241	**0.950**
		iir_biquad_one_section	122	122	**1.000**
		iir_biquad_N_sections	1,054	881	**0.836**
		lms	1,502	1,406	**0.936**
		matrix	37,012	35,223	**0.952**
		matrix_1x3	262	252	**0.962**
		n_complex_updates	2,933	2,885	**0.984**
		n_real_updates	1,570	1,426	**0.908**
		real_update	79	73	**0.924**
	Other	bubblesort	5,086,184	4,969,307	**0.977**
		findmax	19,991	19,990	**1.000**
		keysearch	11,247	11,079	**0.985**
		summidall	19,511	18,514	**0.949**
		summinmax	23,011	23,009	**1.000**
		sumnegpos	18,032	17,035	**0.945**
		sumoddeven	14,783	14,785	**1.000**
		sumposclrneg	28,496	27,467	**0.965**
		sym	107	104	**0.972**
		unweight	340,577	330,378	**0.970**
small average			244,085	239,305	**0.955**
Larger		bitcnt	40,416	38,416	**0.951**
		diskrep	10,028	10,068	**1.004**
		fft	76,505	60,858	**0.795**
		fire	8,900	8,900	**1.000**
		sha	691,047	675,049	**0.977**
		stringsearch	148,849	147,154	**0.989**
larger average			162,624	156,741	**0.953**
average			203,354	197,523	**0.954**

The effect on ACET by performing WCET reduction is shown in Table 8.4. The ACET cycles are obtained from the simulator when random numbers are used as the input data. The ACET in Table 8.4 is equal to the WCET in Table 8.3 for *DSPstone* benchmarks since there is only one path in these benchmarks. In addition, it is very difficult to obtain the WC input data for the *larger* benchmarks, so the same random numbers are used as the input data for these *larger* benchmarks in both Table 8.3 and Table 8.4. Therefore, only *small other* non-*DSPStone* benchmarks have different observed ACET and observed WCET. Compared with the batch sequence, the best sequence to improve WCET found by the genetic algorithm reduces the average ACET for these benchmarks as well. Although the average benefit to ACET (4.5%) is almost the same as the benefit to WCET (4.8%), the ACET of a benchmark can increase when tuning for WCET. For instance, the ACET for *sumoddeven* gets slightly worse. The random input data for this benchmark makes the loop break earlier than WC input data and each path is randomly selected to be executed. This happens to increase the ACET of this benchmark slightly.

In addition to improving WCET, it would be interesting to see the improvement in code size. Table 8.5 shows the results obtained for each benchmark by applying the genetic algorithm when changing the fitness criteria. For each benchmark, three different searches were performed based on *WCET only* (optimizing for WCET), *code size only* (optimizing for space), and *both* (50% for each factor). For each type of search, the effects on both WCET and code size are shown. The results that are supposed to improve according to the specified fitness criteria used are shown in boldface. For these results, the genetic algorithm was able to typically find a sequence for each function that either achieves the same result or obtains an improved result as compared to the batch compilation. When the fitness value is *WCET only*, the overall average WCET is reduced by 5.2% and the overall average code size decreases by 2.2%. When the fitness value is *code size only*, the overall average code size decreases by 7.1%, but the overall average WCET increases by 4.9%. When the fitness value is *both*, the overall average WCET is reduced by 3.5% and the overall average code size decreases by 5.1%. The results when optimizing for both WCET and code size showed that the system is able to achieve a benefit simultaneously on both WCET and code size.

Table 8.5. Comparison of Three Different Fitness Values for the Genetic Algorithm

Category		optimizing for / effect on	WCET Only		Code Size Only		both		
			WCET	Size	WCET	Size	WCET	Size	Avg Both
Small	DSPstone	convolution	**0.950**	0.977	0.950	**0.977**	0.950	0.977	**0.964**
		complex_update	**0.981**	0.970	1.268	**0.964**	0.987	0.941	**0.964**
		dot_product	**0.906**	0.925	0.906	**0.925**	0.906	0.925	**0.915**
		fir	**0.912**	0.980	1.201	**0.933**	0.913	0.953	**0.933**
		fir2dim	**0.951**	0.976	0.949	**0.941**	0.951	0.976	**0.963**
		iir_biquad_one_section	**1.000**	0.993	1.000	**0.993**	1.000	0.993	**0.997**
		iir_biquad_N_sections	**0.838**	0.982	1.000	**1.000**	1.247	1.005	**1.126**
		lms	**0.937**	0.975	0.937	**0.970**	0.849	0.954	**0.901**
		matrix	**0.952**	0.995	1.048	**0.995**	0.968	0.968	**0.968**
		matrix_1x3	**0.957**	0.843	0.957	**0.843**	0.957	0.843	**0.900**
		n_complex_updates	**0.984**	0.986	1.027	**0.966**	0.984	0.986	**0.985**
		n_real_updates	**0.909**	0.967	0.909	**0.967**	0.909	0.967	**0.938**
		real_update	**0.906**	0.941	1.035	**1.015**	0.953	1.015	**0.984**
	Other	bubblesort	**0.964**	0.936	1.033	**0.904**	0.984	0.912	**0.948**
		findmax	**1.000**	0.879	1.000	**0.879**	1.000	0.983	**0.991**
		keysearch	**0.984**	1.143	1.362	**0.672**	1.000	1.058	**1.029**
		summidall	**0.949**	1.018	1.000	**1.000**	1.000	1.071	**1.036**
		summinmax	**1.000**	0.968	1.434	**0.855**	1.000	0.871	**0.935**
		sumnegpos	**0.950**	1.133	1.300	**1.067**	1.000	1.000	**1.000**
		sumoddeven	**0.957**	1.115	1.001	**0.974**	0.979	1.013	**0.996**
		sumposclr	**0.968**	0.889	1.258	**0.877**	1.258	0.877	**1.067**
		sym	**0.978**	0.918	1.000	**0.979**	1.000	0.969	**0.985**
		unweight	**0.971**	0.899	0.971	**0.899**	0.971	0.899	**0.935**
small average			**0.952**	0.974	1.067	**0.939**	0.990	0.963	**0.976**
Larger		bitcnt	**0.911**	1.045	0.945	**0.986**	0.940	0.989	**0.965**
		diskrep	**0.994**	0.929	1.055	**0.786**	1.005	0.818	**0.912**
		fft	**0.800**	0.972	0.852	**0.910**	0.774	0.924	**0.849**
		fire	**1.000**	1.000	1.002	**0.996**	1.002	0.996	**0.999**
		sha	**0.976**	0.989	1.324	**0.903**	0.938	0.939	**0.939**
		stringsearch	**0.978**	0.948	1.005	**0.915**	0.984	0.933	**0.959**
larger average			**0.943**	0.980	1.031	**0.916**	0.941	0.933	**0.937**
overall average			**0.948**	0.977	1.049	**0.928**	0.965	0.948	**0.957**

8.6 Conclusions

There are several contributions that have been presented in this chapter. First, it has been demonstrated that it is possible to integrate a timing analyzer with a compiler and that these WCET predictions can be used by the application developer and the compiler to make phase ordering decisions. Displaying the improvement in WCET during the tuning process allows a developer to interactively tune an embedded application based on WCET feedback. Therefore, the developer can easily gauge the progress that has been made. To the best of our knowledge, this is the first compiler that interacts with a timing analyzer to use WCET predictions during the compilation of applications. Second, it has been shown that WCET predictions can be used as a fitness criteria by a genetic algorithm that finds effective optimization sequences to improve the WCET of applications on an embedded processor. One advantage of using WCET as a fitness criteria is that the searches for an effective sequence are much faster. The development environment for many embedded systems is different than the target environment. Thus, simulators are used when testing an embedded application. Executing the timing analyzer typically requires a small fraction of the time that would be required to simulate the execution of the application. Finally, experimental results indicate that both WCET and code size improvements can be simultaneously obtained when the fitness value of the genetic algorithm addresses both factors. Both of these criteria are important factors when tuning applications for an embedded processor. Note that the estimated WCET can be reduced even though the programs may be too complex to accurately determine the WC input data.

CHAPTER 9

WCET CODE POSITIONING

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

The approach discussed in this chapter is to improve the WCET of an application by applying a WCET code positioning algorithm that searches for the best layout of the code in memory for WCET [6]. Traditional code positioning algorithms are not guaranteed to reduce the WCET of an application since the most frequently executed edges in a program may not be contained in the WC paths. Even if WCET path information were used to drive the code positioning algorithm, a change in the positioning may result in a different path becoming the WC path in a loop or a function. For example, a typical CFG for an *if-then-else* statement is shown in Figure 9.1 with two paths, $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$. Before code positioning (Figure 9.1(a)), assume that the path $1 \rightarrow 3 \rightarrow 4$ is the worst-case path. After block 3 is moved physically in memory next to block 1 to remove the branch penalty from block 1 to block 3 (Figure 9.1(b)), path $1 \rightarrow 2 \rightarrow 4$ may become the new worst-case path since there is a new transfer-of-control penalty from block 1 to block 2. Therefore, a change in the positioning may result in a different path becoming the WC path in a loop or a function. In fact, the new WC path in Figure 9.1(b) may have a higher WCET than the original WC in Figure 9.1(a). In contrast, the frequencies of the edges based on profile data, which are used in traditional code positioning, does not change regardless of how the basic blocks are ordered. Thus, WCET code positioning is inherently a more challenging problem than ACET code positioning.

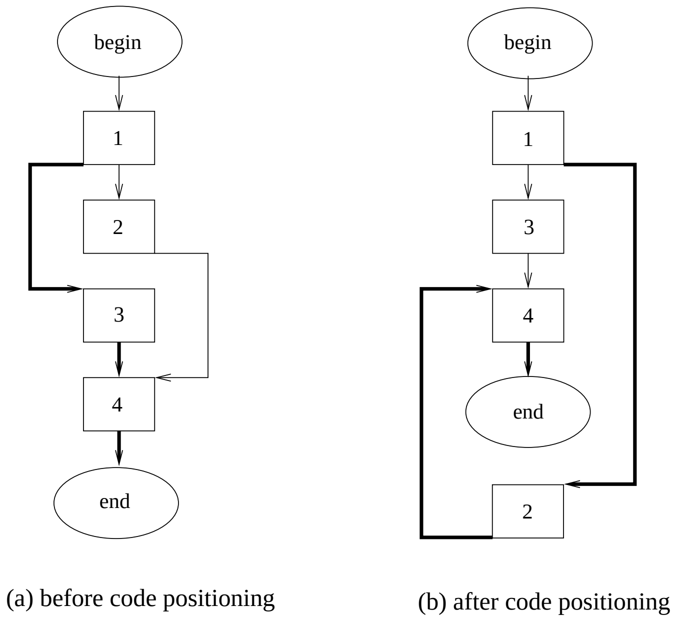


Figure 9.1. Code Positioning for an *if-then-else* Statement

The remainder of this chapter has the following organization. Prior research on code positioning is first summarized. The WCET code position algorithm is then described and an example is used to illustrate the algorithm. WCET target alignment after code positioning is also discussed. Finally, the experimental results to perform the code positioning and target alignment are given.

9.1 Prior Work on Code Positioning

There have been several code positioning (basic block reordering) approaches that have been developed. Pettis and Hansen [64] used execution profile data to position the code in memory. Profile data was used to count the execution frequency for each edge in the CFG. The nodes in the CFG linked by the edges (transitions) with the highest frequency were identified as chains and were positioned contiguously to improve instruction cache performance. Other algorithms have been developed with the primary goal of reducing the number of dynamic transfers of control (e.g. unconditional jumps and taken branches) and the associated pipeline penalty on specific processors. McFarling and Hennessy [65] described a number of code positioning methods to reduce branch misprediction and instruction fetch penalties. Calder and Grunwald [66] proposed an improved code positioning algorithm using a cost model to evaluate the cost of different basic block orderings. They assigned different cost values for different types of transfer-of-control penalties so that they can attempt to select the ordering of basic blocks with the minimal cost. All of these approaches use profile information to obtain a weight for each directed edge between nodes of a CFG by counting the number of times the edge was traversed at run-time. Typical input data instead of WC input data is used in profiling. Thus, these approaches attempt to improve the ACET. In contrast, our code positioning algorithm improves the WCET based on the WCET path information from the timing analyzer.

ACET code positioning techniques rely only on edge frequencies. These frequencies do not change regardless of how the code is positioned. These techniques do not have to perform path analysis. WCET code positioning has to perform path analysis. Since the WCET path may change after changing the positioning, WCET code positioning is more complex than ACET code positioning.

9.2 WCET Code Positioning

Code positioning is essentially an attempt to find the most efficient permutation of the order of basic blocks in a function. Exhaustive approaches are not typically feasible except when the number of blocks is small since there are $n!$ possible permutations, where n is the number of basic blocks in the function. Thus, most approaches use a greedy algorithm to avoid excessive increases in compilation time.

The WCET code positioning algorithm selects edges between blocks to be contiguous in an attempt to minimize the WCET. A directed edge connecting two basic blocks is *contiguous* if the source block is immediately followed by the target block in memory. Making two blocks contiguous can eliminate the transfer-of-control penalty for the transition. However, not all edges can be contiguous. Consider the portion of a control-flow graph shown in Figure 9.2. If edge b (shown as a solid line) is selected to be contiguous, then no other edges to the same target can be contiguous. For example, edge a can no longer be contiguous since its source block 4 cannot be positioned immediately before its target block 2. Likewise, only a single edge among the set that share the same source block can be contiguous. For instance, selecting edge b to be contiguous will make edge c noncontiguous since the target block 3 cannot be positioned immediately after source block 1.

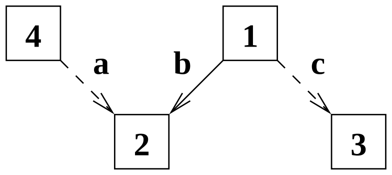


Figure 9.2. Selecting an Edge to Be Contiguous

WCET code positioning needs to be driven by WCET path information. The compiler obtains the WCET and the basic block list for each path in the function from the timing analyzer. If the timing analyzer calculates the WCET path information on the original positioned code, then changing the order of the basic blocks may result in unanticipated increases in the WCET for other paths since previously contiguous edges may become noncontiguous. It was decided instead to treat the basic blocks as being initially unpositioned. Thus, the code is actually modified 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.

The basic idea of this WCET code positioning algorithm is to find the edge that contributes the most to the WCET, which is called the worst-case edge, and make the two basic blocks linked by that edge contiguous to reduce the execution time along the worst-case path. This operation may result in a new worst-case path, so the algorithm positions one edge at a time and re-calculates the new WCET of each path to guide the selection of the next edge to reduce the WCET. At each step, the algorithm attempts to choose the worst-case edge among all the edges along the worst-case paths. Eliminating the transition penalty at the chosen edge will reduce the execution time along the worst-case path and will reduce the execution times along other paths containing this edge as well. However, making one edge contiguous will make other edges noncontiguous.

There are a few terms that need to be defined before the WCET code positioning algorithm can be presented. Edges are denoted as being contiguous, noncontiguous, or unpositioned. A *contiguous* edge has its source block immediately positioned before its target block in memory. In contrast, a *noncontiguous* edge does not. An *unpositioned* edge means that it has not yet been determined if it will be contiguous or noncontiguous. The *upper bound* WCET (UB-WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be noncontiguous. The *lower bound* WCET (LB-WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be contiguous. By selecting an edge to be contiguous, the UB-WCET of the paths containing the edge will decrease and the LB-WCET of some other paths will increase since some *unpositioned* edges become *non-contiguous*. The *weighted* WCET for a path is the WCET for a single iteration of a path multiplied by the possible number of iterations that path will be executed. Paths are also classified as *contributing* or *noncontributing* to the WCET. A path is considered *noncontributing* when its UB-WCET is less than the LB-WCET of another path within the same loop (or outermost level of a function). *Noncontributing* paths cannot affect the WCET. The WCET code positioning algorithm is described in Figure 9.3.

At this point target misalignment penalties are not assessed by the timing analyzer since WCET target alignment, described in Chapter 3, is performed after WCET code positioning. The algorithm selects one *unpositioned* edge at a time to make *contiguous*. An edge is selected by first examining the paths that most affect the WCET. Thus, paths are weighted by the maximum number of times that they can be executed in the function to ensure its effect on

```
WHILE (all the edges in current function have not been positioned){

   FOR (all the paths in the current function) {
      calculate weighted Upper-Bound WCET (UB_WCET), and weighted
      Lower-Bound WCET (LB_WCET) for each path;
   }
   sort the paths ( p1, p2, ..., pi ) in descending order based on
   contributing, weighted Upper-Bound WCET (UB_WCET), and weighted
   Lower-Bound WCET (LB_WCET).
   choose the first contributing path p with at least one
   unpositioned edge in the sorted path list (p1, p2, ..., pi);

   /* choose the best_edge in the path */
   max = -1;
   FOR (each unpositioned edge e in path p ){
      n=0;
      FOR (each path p in the sorted path list(p1,p2, ..., pi)){
         IF (edge e is in path p)
            n++;
         ELSE
            BREAK;
      }
      IF (n>max) {
         max = n;
         best_edge = e
      }
   }

   mark best_edge as contiguous;
   mark the edges that become non-contiguous;
   remove a path from the path list if all its edges have been
   positioned;

}
```

Figure 9.3. The Pseudocode for the WCET Code Positioning Algorithm the WCET is accurately represented. In fact, the number of iterations in which a path may be executed can be restricted due to constraints on branches.

After selecting an edge to be *contiguous* (and possibly making one or more other edges *noncontiguous*), the UB-WCET and LB-WCET of each path are recalculated. By making two blocks contiguous, a useless jump inserted at the beginning of the algorithm will be removed or a taken branch may become a non-taken branch between the two contiguous blocks. At the same time, the branch condition may be reversed to reflect the change. The UB-WCET will decrease step by step since more and more edges are made contiguous while the LB-WCET will increase since more and more *unpositioned* edges become *noncontiguous*. The algorithm continues until all edges have been positioned. At the end of the algorithm, the LB-WCET and UB-WCET should be the same for every path.

Consider the source code in the Figure 9.4, which is a contrived example to illustrate the algorithm. Figure 9.5 shows the corresponding control flow that is generated by the compiler. While the control flow in the figure is represented at the source code level to simplify its presentation, the analysis is performed by the compiler at the assembly instruction level after compiler optimizations are applied to allow more accurate timing predictions. Note that some branches in Figure 9.5 have conditions that are reversed from the source code to depict the branch conditions that are represented at the assembly instruction level. Several unconditional jumps, represented in Figure 9.5 as ***goto*** statements underneath dashed lines, have been inserted to make all transitions between basic blocks result in a transfer of control penalty. The unconditional jumps in blocks 3 and 6 were already present. Conditional branches are represented as ***if*** statements in Figure 9.5. The jumps (shown as ***goto*** statements) immediately following each conditional branch are actually placed in separate basic blocks within the compiler's representation, but are shown in the same block as the corresponding branch in the figure to simplify the presentation of the example. The transitions (directed edges) between nodes are labeled so they can be referenced later. Figure 9.6 shows the paths through the control flow graph. Paths A-D represent paths within the loop. Path E represents the outer level path, where the loop is considered a single node within that path. Backedges (directed edges back to the entry point of the loop) are considered to be part of the paths within the loop since these edges can be traversed on all loop iterations, except for the last one. Likewise, the exit edges (directed edges leaving

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

Figure 9.4. An Example to Illustrate the Algorithm

the loop) are considered part of the outer paths containing the loop since an exit edge is executed at most once each time the loop is entered.

Table 9.1 shows how WCET code positioning is accomplished for the example shown in Figures 9.5 and 9.6. At each step the status for each edge and the current UB-WCET and LB-WCET for each path calculated from the timing analyzer are shown. Initially all edges are unpositioned, as shown in step 0. For each step an edge is selected to be *contiguous* and one or more edges become *noncontiguous*. Thus, after each step one or more paths have their UB-WCET reduced and one or more paths have their LB-WCET increased. At the first step, the algorithm selects edge j to be *contiguous* since it reduces the UB-WCET of all four paths in the loop. This selection also causes edges a and k to become *noncontiguous*, which results in only a small increase for the LB-WCET of the entire function (path E) since these edges are outside the loop. In the second step, edge i is selected since it is part of path D, which contains the greatest current UB-WCET. The algorithm chooses edge i instead of another edge in path D since edge i is also part of path B, which contains the second greatest WCET at that point. Since path B and path D share edge i, the UB-WCET of the two paths decreases by 3 cycles. By making edge i contiguous, edge h becomes *noncontiguous*. Both

Path A and Path C contain the *noncontiguous* edge h, so the LB-WCET of both paths A and C increases by 3 cycles. Edge g is selected to be *contiguous* in the third step since that is also part of path D, which still contains the greatest UB-WCET. The UB-WCET of both path B and path D decreases because edge g is a part of both paths. The LB-WCET of both path A and path C increases because edge f becomes *noncontiguous* while both path A and path C contain edge f. Edge e becomes *contiguous* in the fourth step since it is part of path C, which currently contains the greatest UB-WCET. And edge c is the only *unpositioned* edge along path C. At this point path D's UB-WCET becomes 29, which is less than the

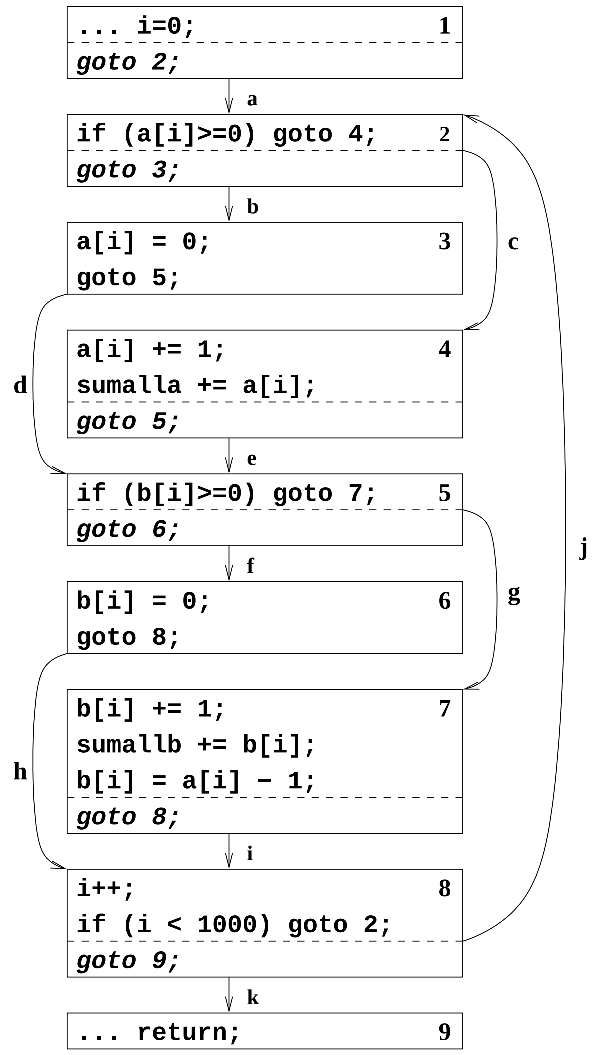


Figure 9.5. Control Flow Graph of Code in the Example

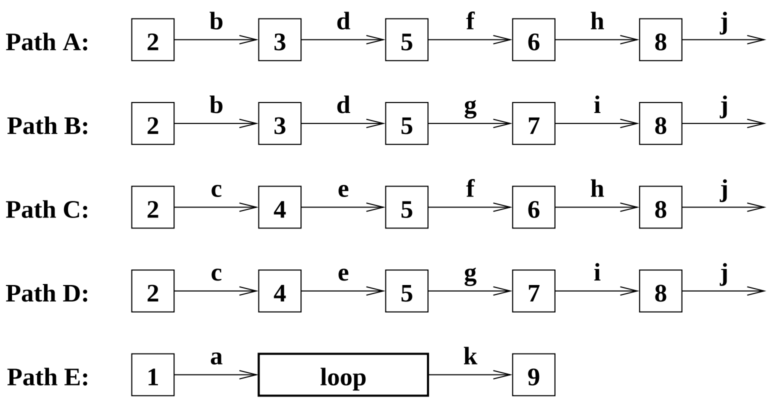


Figure 9.6. Paths of the Example in Figure 9.5

Table 9.1. Code Positioning Steps in the Example

S	Status of Edges											WCETs of Paths Shown in Figure 9.6									
t												UB-WCET					LB-WCET				
p	a	b	c	d	e	f	g	h	i	j	k	A	B	C	D	E	A	B	C	D	E
0	u	u	u	u	u	u	u	u	u	u	u	36	40	37	41	37,020	21	25	22	26	22,018
1	n	u	u	u	u	u	u	u	u	c	n	33	37	34	38	34,024	21	25	22	26	22,024
2	n	u	u	u	u	u	u	n	c	c	n	33	34	34	35	31,024	24	25	25	26	22,024
3	n	u	u	u	u	n	c	n	c	c	n	33	31	34	32	30,024	27	25	28	26	24,024
4	n	u	u	n	c	n	c	n	c	c	n	33	31	31	29	29,024	30	28	28	26	26,024
5	n	c	n	n	c	n	c	n	c	c	n	30	28	31	29	27,024	30	28	31	29	27,024
u = unpositioned, c = contiguous, n = noncontiguous																					

LB-WCET of 30 for path A. Thus, path D is now *noncontributing.* During the fifth step edge b is selected since it is part of path A, which contains the current greatest UB-WCET. At this point all of the edges have been positioned and the UB-WCET and LB-WCET for each path are now identical. The original positioning shown in Figure 9.5, but without the extra jumps inserted to make all transitions noncontiguous, has a WCET of 31,018 or about 14.8% greater than after WCET code positioning.

While the edges have been positioned according to the selections shown in Table 9.1, the final positioning of the basic blocks still has to be performed. The list of contiguous edges in the order in which they were selected are $8 \rightarrow 2, 7 \rightarrow 8, 5 \rightarrow 7, 4 \rightarrow 5$, and $2 \rightarrow 3$. Six of the nine blocks are positioned in the order $4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 2 \rightarrow 3$ by connecting these edges by their common nodes. The remaining blocks, which are 1, 6, and 9, can be placed either before or after this contiguous set of blocks. In general, there may be several contiguous sets of blocks in a function and these sets can be placed in an arbitrary order. The entry block of the function is always designated as the first block in the final positioning to simplify the generation of the assembly code by the compiler and the processing by the timing analyzer. Note that the entry block can never be the target of an edge in the control flow due to prologue code for the function being generated in this block. The process of the code positioning for the example is summarized in Figure 9.7, where the contiguous edges have thicker transitions and the steps are identified in which the edges are positioned.

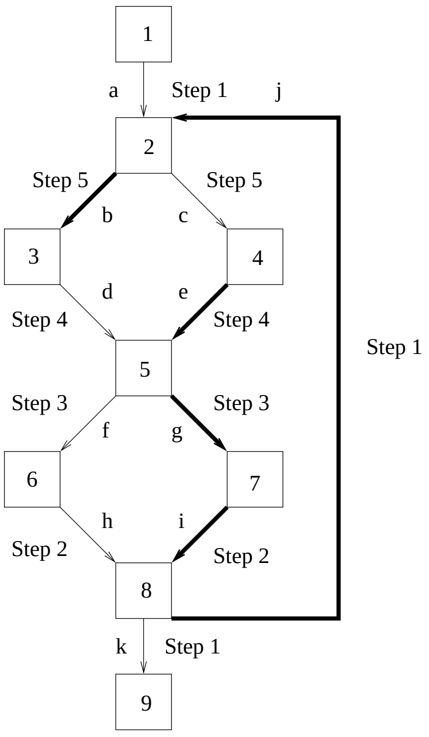


Figure 9.7. The Steps to Make Edges Contiguous

Figure 9.8 shows the final positioning of the code after applying the WCET code positioning algorithm. By contrasting the code in Figure 9.5 with the final positioning in Figure 9.8, one can observe that performing the final positioning sometimes requires reversing branch conditions, changing target labels of branches, labeling blocks that are now targets of branches or jumps, inserting new unconditional jumps, and deleting other jumps. All of the loop paths A-D required three transfers of control prior to WCET code positioning. After WCET code positioning paths A and C each require three transfers of control and paths B and D each require only one. Note that paths B and D had higher UB-WCETs before the edges were positioned.

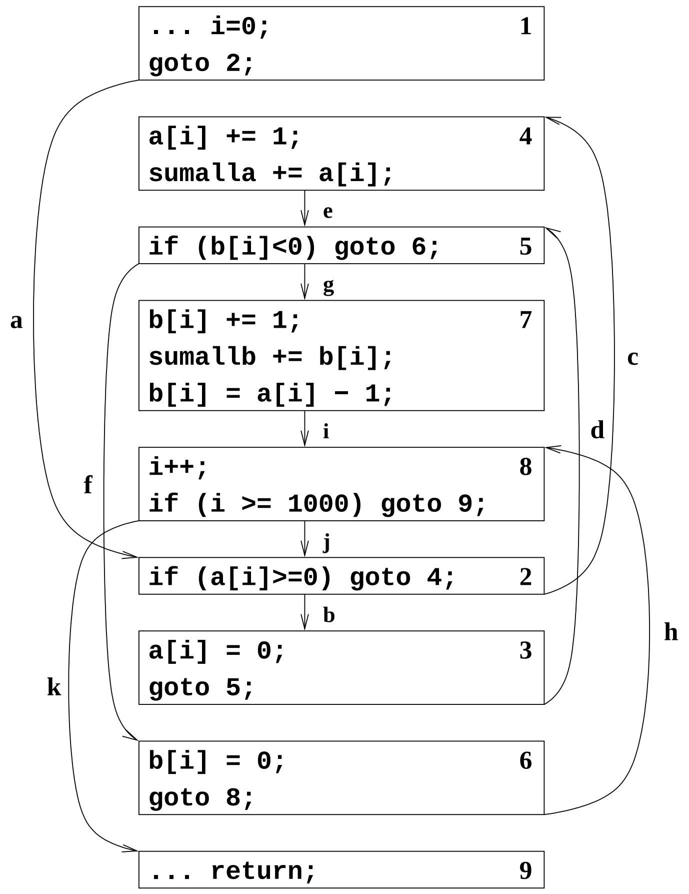


Figure 9.8. The Example in Figure 9.5 after WCET Positioning

The portion of the greedy algorithm (shown in Figure 9.3) that most affects the analysis time is the computation performed by the timing analyzer, which is invoked each time an edge is selected to become contiguous. Given that there are n basic blocks in a function, there can be at most n-1 contiguous edges and sometimes there are less. For instance, only five edges were selected to be contiguous instead of n-1 or eight edges for the example shown

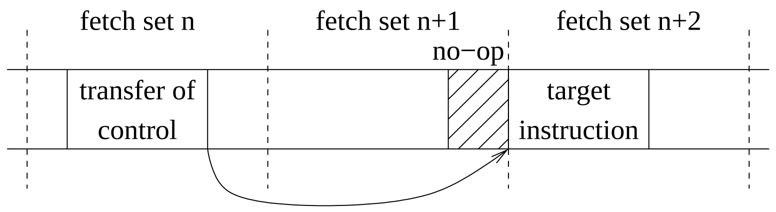


Figure 9.9. Aligning a Target Instruction

in Table 9.1 and Figure 9.8. Thus, the timing analyzer is invoked at most n-1 times for each function, which is much less than the $n!$ invocations that would be required if every possible basic block ordering permutation was checked.

Code positioning is performed after all other optimizations that can affect the instructions generated for a function. This includes inserting instructions to manage the activation record on the run time stack and instruction scheduling. The compiler can invoke the timing analyzer at each step of the code positioning since all required transformations have been performed on the code. After code positioning, the branch target misalignment optimization is performed.

9.3 WCET Target Alignment

After the basic blocks have been positioned within a function, WCET target alignment is performed to further reduce the extra transfer of control penalties due to misaligned targets (ref. Chapter 3). No-ops are added before the target instruction to make the target instruction fit into one fetch set. Figure 9.9 shows an example where the target instruction is in a single fetch set after adding a no-op instruction to force this misaligned target instruction to not span the fetch set boundary (compared to Figure 3.1).

WCET target alignment attempts to minimize the number of target misalignment penalties in the following manner. First, in order to find the right place to add no-op instructions, the function is partitioned into relocatable sets of basic blocks. The first block in a relocatable set is not fallen into from a predecessor block and the last block ends with an unconditional transfer of control, such as an unconditional jump or a return. A relocatable set of blocks can be moved without requiring the insertion of additional instructions. For instance, the code in Figure 9.8 after WCET positioning has four relocatable sets of blocks, which are {1}, {4,5,7,8,2,3}, {6}, and {9}. In contrast, the original flow graph of blocks in Figure 9.5 has three relocatable sets, which are {1,2,3}, {4,5,6}, and {7,8,9}. After WCET code positioning, the relocatable sets of blocks are aligned one set at a time from the top of the function to the bottom of the function by inserting no-ops before relocatable sets. Since each instruction has to be aligned on a word boundary (2 bytes) and each fetch set consists of 4 words, there are four different possible positionings for each relocatable set. The different alignments are accomplished by inserting 0, 1, 2, or 3 no-ops before the beginning of the relocatable set, where each no-op instruction is one word in size. The locations where no-ops can be inserted before each relocatable set of blocks is illustrated in Figure 9.10. Note that these no-ops instructions are not reachable in the control flow and are never executed. The timing analyzer is invoked four times to determine the best number of inserted no-ops (from 0 to 3) for each relocatable set of blocks based upon the WCET of the function. Thus, the timing analyzer is invoked 4(m-1) times for each function, where m is the number of relocatable sets of blocks to be aligned. The best number of no-ops with the lowest WCET for the function is chosen for each relocatable set. In the case that the WCET is the same for two or more options, the option with the fewest no-ops is selected. To help support this analysis, an option was added to the timing analyzer to only assess misalignment penalties within a range of blocks. Therefore, when the best number of no-ops is determined for a relocatable set at the top of the function, the effect of these no-ops on the remaining relocatable sets not yet aligned is not considered since these relocatable sets at the bottom of the function will be aligned later anyway.

A more aggressive approach could be attempted by trying all permutations of ordering relocatable sets of blocks in addition to inserting no-ops. This approach could potentially reduce the number of no-ops inserted. However, the code size increase is small and the current approach is quite efficient.

9.4 Results

This section describes the results of a set of experiments to illustrate the effectiveness of improving the WCET by using WCET code positioning and WCET target alignment.

Table 9.2 shows the benchmarks and applications used to test these WCET reduction algorithms. Note that the code positioning algorithm is based upon the worst-case path information, while most *DSPStone* benchmarks do not have conditional constructs, such as if statements, which means they have only one path. Therefore, the *DSPStone* benchmarks are not used for the experiments on WCET code positioning. Other benchmarks were selected since they do have conditional constructs, which means the WCET and ACET input data may not be the same.

In Table 9.2, the *base* compilation time is the time without performing code positioning, while the *position* compilation time is the time with WCET code positioning and WCET target alignment. The *time ratio* indicates the compilation overhead of performing WCET code positioning and target alignment. Most of this overhead is due to repeated calls to the timing analyzer. While this overhead is reasonable, it could be significantly reduced if the timing analyzer and the compiler were in the same executable and passed information via

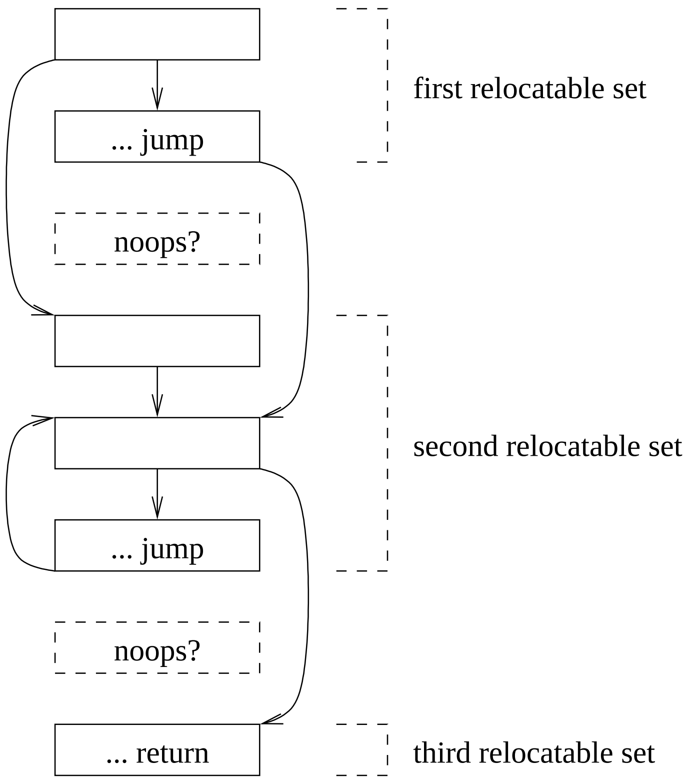


Figure 9.10. Inserting No-op Instructions Before Relocatable Set of Blocks

Table 9.2. The Compilation Time of the Benchmarks

Category	Benchmarks	Compilation Time		time
		base(min)	position(min)	Ratio
Small	bubblesort	0.23	0.37	1.609
	findmax	0.97	0.97	1.000
	keysearch	0.18	0.22	1.222
	summidall	0.13	0.13	1.000
	summinmax	0.13	0.17	1.308
	sumnegpos	0.13	0.13	1.000
	sumoddeven	0.15	0.17	1.133
	sumposclr	0.15	0.20	1.333
	sym	0.18	0.18	1.000
	unweight	0.13	0.13	1.000
small average		0.24	0.27	1.161
Larger	bitcnt	0.32	0.37	1.156
	diskrep	0.22	0.50	2.273
	fft	0.28	0.70	2.500
	fire	0.17	0.22	1.294
	sha	0.42	0.87	2.425
	stringsearch	0.40	0.97	1.125
larger average		0.30	0.61	1.953
overall average		0.26	0.39	1.458

arguments instead of files. Note that *base* compilation time in the table is slightly longer than the regular VPO compilation time since the compiler has to invoke the timing analyzer at the end of the compilation to obtain the baseline WCET.

Table 9.3 shows the effect on WCET after code positioning and target alignment. The results *before positioning* indicate the measurements taken after all optimizations have been applied except for WCET code positioning and WCET target alignment. The *observed cycles* were obtained from running the compiled programs with WC input data through the SC100 simulator. The *WCET cycles* are the WCET predictions obtained from the timing analyzer. The *WCET ratios* show that these predictions are reasonably close for *Small* programs, but are much larger than the observed cycles for *larger* benchmarks. The *observed cycles* after

Table 9.3. WCET Results after WCET Code Positioning and Target Alignment

Program	Before Positioning			After Positioning		After Alignment	
	Observed Cycles	WCET Cycles	WCET Ratio	WCET Cycles	Positioning Ratio	WCET Cycles	Alignment Ratio
bubblesort	7,365,282	7,616,292	1.034	7,614,792	1.000	7,490,043	0.983
findmax	19,997	20,002	1.000	19,009	0.950	19,009	0.950
keysearch	30,667	31,142	1.015	29,267	0.940	29,267	0.940
summidall	19,513	19,520	1.000	16,726	0.857	16,726	0.857
summinmax	23,009	23,015	1.000	21,021	0.913	20,021	0.870
sumnegpos	20,010	20,015	1.000	18,021	0.900	18,021	0.900
sumoddeven	22,025	23,032	1.046	18,035	0.783	16,546	0.718
sumposclrneg	31,013	31,018	1.000	27,024	0.871	27,024	0.871
sym	55,343	55,497	1.003	51,822	0.934	51,822	0.934
unweight	350,507	350,814	1.001	321,020	0.915	321,020	0.915
Small average	793,737	819,035	1.010	813,674	0.906	800,950	0.894
bitcnt	39,616	55,620	1.404	52,420	0.942	52,321	0.941
diskrep	9,957	12,494	1.255	11,921	0.954	11,907	0.953
fft	73,766	73,834	1.001	73,776	0.999	73,778	0.999
fire	8,813	10,210	1.159	10,210	1.000	10,210	1.000
sha	691,045	769,493	1.114	769,461	1.000	759,179	0.987
stringsearch	147,508	194,509	1.319	186,358	0.958	186,304	0.958
larger average	161,784	186,027	1.208	184,024	0.976	182,283	0.973
overall average	556,754	581,657	1.084	577,555	0.932	569,950	0.924

WCET positioning or WCET alignment were not obtained since this would require new WCET input data due to changes in the WCET paths.

The results *after positioning* indicate the measurements taken after the positioning algorithm described in Section 9.2 is applied immediately following the preceding optimization phases. The *WCET cycles* represent the new predicted WCET by the timing analyzer. The *positioning ratio* indicates the ratio of the *WCET cycles after positioning* divided by the *WCET cycles before positioning.* There was over a 9% average reduction in WCET for *small* benchmarks by applying the WCET code positioning algorithm, while there was 6.8% average reduction in WCET for all benchmarks. The results *after alignment* indicate the

Table 9.4. ACET Results after WCET Code Positioning and Target Alignment

Program	Baseline	After Positioning		After Alignment	
	ACET Cycles	ACET Cycles	Positioning Ratio	ACET Cycles	Alignment Ratio
bubblesort	5,086,177	5,084,809	1.000	5,024,547	0.988
findmax	19,991	17,020	0.851	17,020	0.851
keysearch	11,067	10,399	0.940	10,399	0.940
summidall	19,511	16,721	0.857	16,721	0.857
summinmax	23,009	20,532	0.892	20,018	0.870
sumnegpos	18,032	15,042	0.834	15,042	0.834
sumoddeven	14,783	10,764	0.728	11,097	0.751
sumposclrneg	28,469	25,561	0.898	25,561	0.898
sym	107	107	1.000	107	1.000
unweight	340,577	311,088	0.913	311,088	0.913
Small average	556,172	551,204	0.891	545,160	0.890
bitcnt	39,616	37,516	0.947	37,417	0.944
diskrep	9,957	9,486	0.953	9,568	0.961
fft	73,766	73,714	0.999	73,714	0.999
fire	8,813	8,813	1.000	8,813	1.000
sha	691,045	691,048	1.000	683,051	0.988
stringsearch	147,508	147,510	1.000	147,455	1.000
larger average	161,784	161,348	0.983	160,003	0.982
overall average	409,277	405,008	0.926	400,726	0.925

measurements that were obtained after the WCET target alignment algorithm in Section 9.3 is applied following WCET code positioning. The *WCET cycles* again represent the new predicted WCET by the timing analyzer. The *alignment ratio* indicates the ratio of the *WCET cycles after alignment* as compared to the *WCET cycles before positioning.* Three of the ten *Small* benchmarks improved due to WCET target alignment, while three out of six *Larger* benchmarks improved, which resulted in over an additional 0.8% average reduction in WCET.

The effect on ACET after WCET code positioning is shown in Table 9.4. The ACET cycles are obtained from the simulator when random numbers are used as the input data.

The baseline ACET cycles are obtained before code positioning. The average ACET for these benchmarks after code positioning is reduced by 7.4%. Although the goal of the target alignment is also for WCET, it also reduces ACET by 0.1%. While some benchmarks get similar ACET benefits as WCET benefits, such as benchmarks *keysearch* and *summidall*, some other benchmarks have ACET benefits that are greater or less than the WCET benefits. Since the code positioning algorithm reduces the execution time of the WC paths while increasing the execution time of other paths, the ACET benefit after code positioning depends on how frequently the WC paths are driven by the ACET input data. Furthermore, if the blocks comprising the frequent path are a subset of the blocks comprising the WC path, WCET code positioning may reduce the WCET while not increasing the execution time of other paths. For instance, the difference in the execution time of the two paths in the benchmark *findmax* is only 1 cycle. After code positioning, the execution time of the original WC path is reduced by 3 cycles while the execution time of the other path stays the same. Therefore, the other path becomes the new WC path. The WCET is reduced by only 1 cycle each iteration since the WC path changes. However, the ACET is obtained by using random input data, which drives both paths. Since the execution time of one path is reduced by 3 cycles and the baseline in cycles for ACET is smaller than the WCET baseline, the ACET benefit is larger than the WCET benefit after code positioning for this benchmark.

While the results in Table 9.3 show a significant improvement in the predicted WCET, it would be informative to know if better positionings than those obtained by the greedy WCET code positioning algorithm are possible. The functions in these *Small* benchmarks were small enough so that the WCET for every possible permutation of the basic block ordering could be estimated. The number of possible orderings for each function is $n!$, where n is the number of basic blocks, since each block can be represented at most once in the ordering. Table 9.5 shows the results of performing an exhaustive search for the best WCET code positioning for *Small* benchmarks, where the WCET is calculated for each possible permutation. Unlike the measurements shown in Table 9.3, these WCET results exclude target misprediction penalties. The WCET positioning algorithm does not take target misprediction penalties into account when making positioning decisions since the WCET target alignment optimization occurs after positioning. Thus, the WCETs in Table 9.5 are in general slightly lower than the WCETs shown in Table 9.3.

Table 9.5. The Code Positioning Algorithm Found the Best Layout for *Small* Benchmarks

Program	Permuta-tions	Minimum WCET	Greedy		Default		Maximum	
			WCET	Ratio	WCET	Ratio	WCET	Ratio
bubblesort	362,883	7,614,792	7,614,792	1.000	7,616,292	1.000	8,990,017	1.181
findmax	120	19,009	19,009	1.000	20,002	1.052	24,999	1.315
keysearch	39,916,801	29,237	29,237	1.000	31,112	1.064	59,574	2.038
summidall	5,040	16,726	16,726	1.000	18,520	1.107	28,722	1.717
summinmax	362,880	20,021	20,021	1.000	23,015	1.150	29,017	1.449
sumnegpos	5,040	18,021	18,021	1.000	20,015	1.111	28,017	1.555
sumoddeven	3,628,800	16,034	16,034	1.000	22,049	1.375	31,054	1.937
sumposclrn	362,880	27,024	27,024	1.000	31,018	1.148	37,020	1.370
sym	5041	51,822	51,822	1.000	55,497	1.071	62,979	1.215
unweight	40,320	321,020	321,020	1.000	350,714	1.092	471,316	1.468
average	4,925,214	814,121	814,121	1.000	818,823	1.117	976,272	1.524

The number of *permutations* varied depending upon the number of *routines* in the benchmark and the number of basic blocks in each function. The *minimum WCET* represents the lowest WCET found by performing the exhaustive search. The *greedy WCET* is the WCET obtained by the greedy code positioning algorithm described in Figure 9.3. The *ratio* is the WCET divided by *minimum WCET*. There are typically multiple code positionings that result in an equal *minimum WCET*. The *greedy WCET* obtained by the code positioning algorithm was always identical to the *minimum WCET* for each function in each benchmark for the *Small* test suite. It appears that the greedy algorithm is very effective at finding an efficient WCET code positioning. The *default WCET* represents the WCET of the default code layout without *WCET code positioning*. On average the *default WCET* is 11.7% worse than the *minimum WCET*. The *maximum WCET* represents the highest WCET found during the exhaustive search. The results show that the *maximum WCET* is 52.4% higher on average than the *minimum WCET*. While the *default WCET* is relatively efficient compared to the *maximum WCET*, the *greedy WCET* still is a significant improvement over just using the default code positioning.

Table 9.6 shows the number of possible permutations for the layout of each function and the number of code layouts with the same minimal WCET found by performing these exhaustive searches. As shown in the table, there is more than one code layout with the same minimal WCET for most of the functions. However, the number of layouts with the same

Table 9.6. The Number of the Best Layouts with Minimal WCET for *Small* Benchmarks

Program	Functions	Permutations	Best Layouts
bubblesort	main	1	1
	initialize	1	1
	exchange	1	1
	bubblesort	362,880	3
findmax	main	120	12
keysearch	main	1	1
	foo	39,916,800	2
summidall	main	5,040	12
summinmax	main	362,880	96
sumnegpos	main	5,040	24
sumoddeven	main	3,628,800	48
sumposclrn	main	362,880	48
sym	main	1	1
	is_sysmetric	5,040	6
unweight	main	40,320	48
average		2,979,320	20

minimal WCET is a very small percentage of the total number of possible permutations. This shows that the greedy algorithm used is quite effective.

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

9.5 Conclusions

There are several contributions that have been presented in this chapter. First, a WCET code positioning algorithm was developed, which is driven by WCET path information from timing analysis, as opposed to ACET frequency data from profiling. The WCET code positioning is inherently more challenging than ACET code positioning since the WC paths may change after updating the order of basic blocks. In contrast, the frequency of the edges based on profile data, which is used in ACET code positioning, does not change regardless of how the basic blocks are ordered. Second, experiments show the greedy WCET code positioning algorithm obtains optimal results on the SC100 for the suite of programs with a small number of basic blocks. Finally, WCET branch target alignment has also been implemented and evaluated. The target alignment optimization reduces WCET due to target misalignment penalties. Thus, it is shown that it is feasible to develop specific compiler optimizations that are designed to improve WCET using WC path information as opposed to improving ACET using frequency data. Code positioning determines the order of the basic blocks, but, in general, it does not change the code size. Therefore, code positioning is an appropriate compiler optimization to reduce the WCET for embedded systems since the space for the code in embedded systems is also a limited resource.

CHAPTER 10

WCET PATH OPTIMIZATION

Traditional frequent path-based compiler optimizations are performed based upon the path frequency information gathered from profiling to reduce ACET. In this chapter, path-based optimizations applied on WC paths to reduce WCET are described [7]. The WC paths within each loop and function are identified by the timing analyzer and the paths are distinguished in the control flow using code duplication. In this way, traditional path-based compiler optimizations, designed for reducing the execution time along the frequently executed paths, can be adapted to reduce the execution time along the worst-case paths. These path-based optimizations reduce the execution time at the expense of the code size increase. Therefore, each path optimization is not committed unless it reduces the WCET.

The remainder of this chapter has the following organization. Prior research on path optimizations is first summarized. The compiler optimization techniques used in this chapter are then described separately. An example is used to illustrate how these optimizations enable other optimizations. Finally, the experimental results show that a WCET reduction versus a code size increase.

10.1 Prior Work on Path Optimizations

There has been a significant amount of work over the past few of decades on developing path 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 [67]. A related technique that was developed later is called superblock formation and scheduling [68]. 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 [69] and conditional branches [70] and to perform partial dead code elimination and partial redundancy elimination [71].

10.2 WC Path Optimizations

The compiler is integrated with the timing analyzer to obtain the WC path information. Therefore, traditional path optimizations used for frequent paths can be applied along the WC paths. Several compiler optimization techniques used in this chapter are described in this section.

10.2.1 WC Superblock Formation

A superblock is a sequence of basic blocks in the 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 10.1 illustrates the WC *superblock formation* process. Figure 10.1(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. WC *superblock formation* starts at the beginning of the WC path and duplicates 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 10.1(b) shows the control flow after duplicating code along the WC path. The superblock consists of blocks 2, 3, 5', 6', 8', which are shown in bold font. 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.

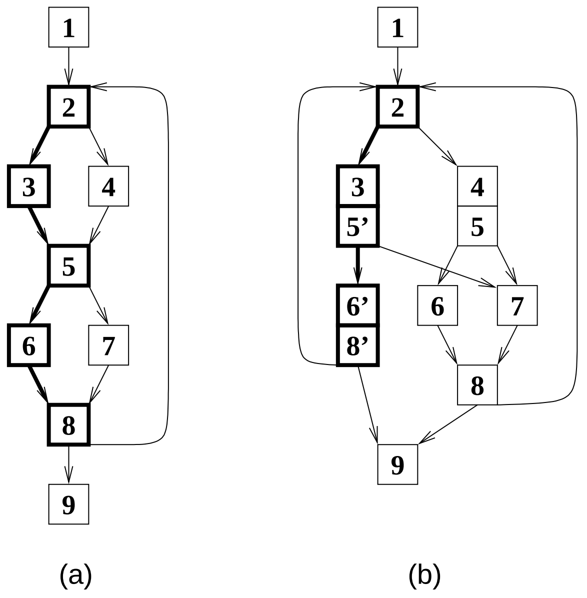


Figure 10.1. Example Illustrating Superblock Formation

10.2.2 WC Path Duplication

Figure 10.2 shows the control-flow graph from Figure 10.1(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, 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 10.1(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, then 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 10.2 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 original loop iterations which can enhance the opportunities for other optimizations.

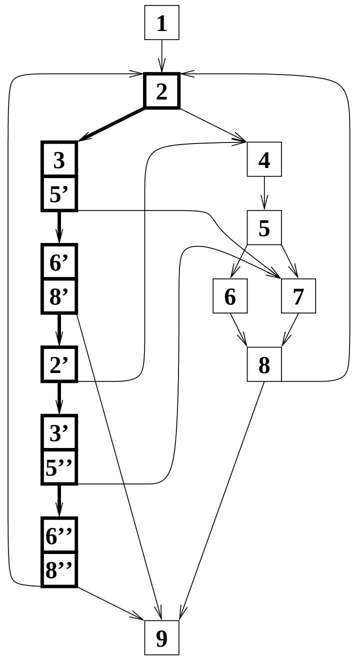


Figure 10.2. WC Path Duplication of Graph in Figure 10.1(b)

WC path duplication presents interesting challenges for the timing analyzer and the compiler since some acyclic paths, such as 2→...→8" in Figure 10.2, represent two iterations of the original loop and others, such as 2→4→...→8, represent a single iteration. The duplicated loop header, block 2' in Figure 4, is annotated so that the timing analyzer counts an extra iteration for any path containing it. The compiler was also modified 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.

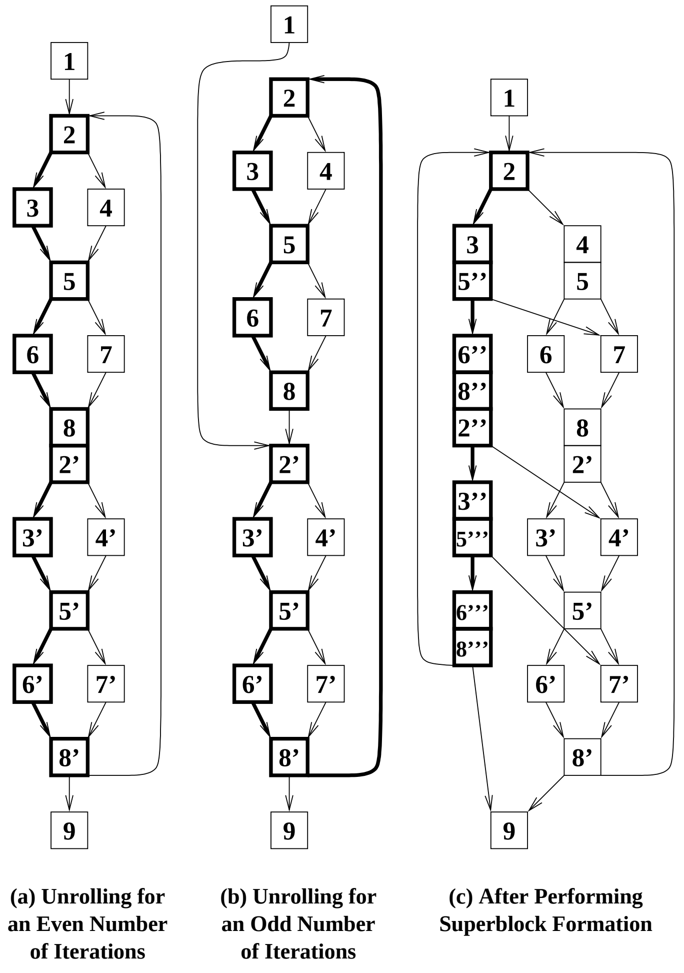
10.2.3 WC Superblock Formation after Loop Unrolling

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. In this dissertation, limited *loop unrolling* is performed 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 the code size increase should be limited. Some approaches that perform unrolling require a cleanup loop to handle exits from the superblock and this cleanup loop can be unstructured. Such an approach was not used 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 10.3(a) shows the control flow from Figure 10.1(a) after unrolling by a factor of two when the original loop had an even number of iterations. Figure 10.3(b) shows how the 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 a 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 10.3(b) starts at block 2', the loop header, and ends at block 8. In both Figure 10.3(a) and Figure 10.3(b) the compare and branch instructions in block 8 are eliminated, reducing both the ACET and WCET. However, the approach in Figure 10.3(b) does not result in any merged blocks, such as blocks 8 and 2' in Figure 10.3(a), which may result in fewer other compiler optimizations being enabled. As illustrated in Figures 10.2 and 10.3, *path duplication* results in less code duplication than *loop unrolling*. However, *loop unrolling* can results in a greater reduction in WCET than *path duplication*. *Superblock formation* after *loop unrolling* results in a larger superblock consisting of code from two iterations (Figure 10.3(c)). Other compiler optimizations can potentially find more opportunities to reduce the execution time along the superblock.

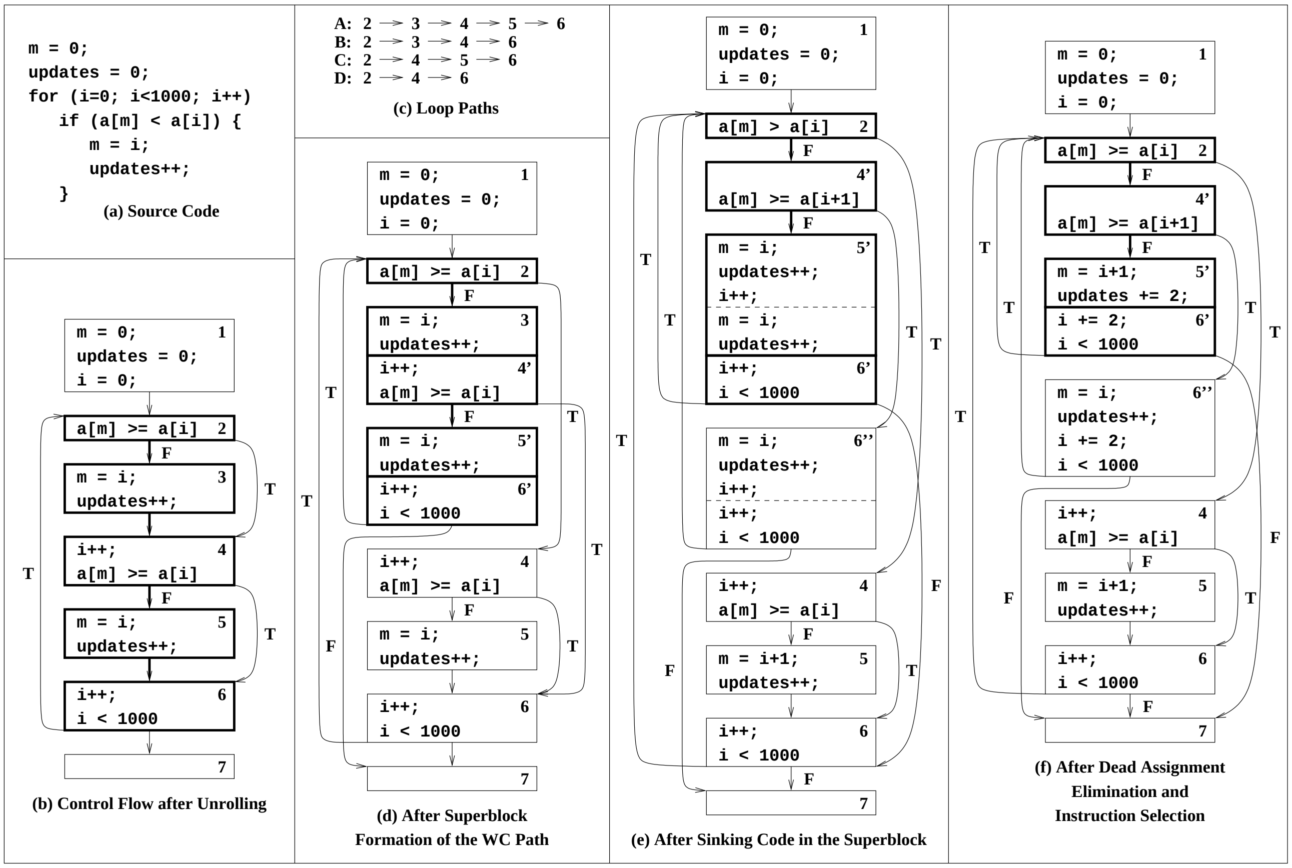
10.3 Enabling Other Optimizations

Superblock formation, *path duplication*, and *loop unrolling* may enable other compiler optimizations. For instance, consider Figure 10.4(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 10.4(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



1
2
3
4
5
6
7
8
2'
3'
4'
5'
6'
7'
8'
9
(a) Unrolling for an Even Number of Iterations
1
2
3
4
5
6
7
8
2'
3'
4'
5'
6'
7'
8'
9
(b) Unrolling for an Odd Number of Iterations
1
2
3
5''
4
5
6''
8''
2''
6
7
8
2'
3''
5'''
3'
4'
5'
6'''
8'''
6'
7'
8'
9
(c) After Performing Superblock Formation

Figure 10.3. Unrolling Followed by Superblock Formation 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.



```
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
m = 0; updates = 0; i = 0; 1
a[m] >= a[i] 2
m = i; updates++; 3
i++; a[m] >= a[i] 4
a[m] >= a[i+1] 4'
m = i; updates++; 5
m = i; updates++; i++; 5'
m = i+1; updates++; 5
m = i+1; updates += 2; 5'
i++; i < 1000 6
i++; i < 1000 6'
i += 2; i < 1000 6'
m = i; updates++; i++; i++; i < 1000 6''
m = i; updates++; i += 2; i < 1000 6''
a[m] > a[i] 2
7
T
F

Figure 10.4. Illustrating WCET Superblock Formation and Associated Optimizations

The compiler obtains the WCET for each path in the function from the timing analyzer. All basic blocks are initially marked unpositioned. This step is the exact same as the first step of the code positioning described in Chapter 9.

Figure 10.4(c) enumerates the four different paths through the loop. Transfer of control penalties are initially assessed between each basic block. 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.

WC path optimizations are attempted on the WC path in the innermost loops of a function or at the outer level if the function has no loops since the innermost loops and functions without no loops are the leaves of the timing tree which contribute most of the execution time. Once the WC path is identified, *superblock formation* is attempted on that path. This means that code is duplicated so that the path is only entered at the head of the loop. Consider Figure 10.4(d), where the 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 also enable other compiler optimizations. In Figure 10.4(d), blocks 3 and 4' are merged together and blocks 5' and 6' are merged together. Removing joins (incoming transitions) from the WC path may also enable some 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 while it can potentially increase the code size. The two assignments in block 3 of Figure 10.4(d) and the increment of i from block 4' in Figure 10.4(d) are sunk after the fallthrough transition of block 4' into the top portion of block 5' in Figure 10.4(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 10.4(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 10.4(e) is now dead since its value is never used and this assignment is deleted in Figure 10.4(f). Likewise, the multiple increments to the *updates* variable in block 5’/6’ of Figure 10.4(e) are combined into a single instruction in block 5’/6’ of Figure 10.4(f). In addition, the two pair of increments of i in blocks 5’/6’ and in block 6” are combined into single increments ”*i += 2;*”. 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 into ”*m = i+1;*”. Other optimizations are also re-applied that can exploit the superblock control flow with its single entry point. These optimizations include constant propagation, copy propagation, and strength reduction.

Some transformations, such as distinguishing the WC path through superblock formation and code sinking, can increase the number of instructions in a function. Since these optimizations are targeted for embedded systems, code size is not increased unless there is a corresponding benefit gained from decreasing the WCET. As mentioned previously, VISTA has the ability to discard previously applied transformations which can be used to roll back transformations when the WCET is not reduced. 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 the state of the program representation prior to the point when the transformation was applied is restored. 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 the resulting transformation will be beneficial.

A sequence of 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 first instruction produces a result that the second instruction cannot access when it is not being available in the pipelines due to no pipeline interlocks. Since these path optimizations may increase the code size, they are not committed unless the WCET is reduced. In order to make this decision, the WCET before and after each path optimization has to be calculated. However, these required phases have to be applied before the WCET can be obtained. Therefore, these required optimizations have to be reversed and/or re-applied, depending on if the optimization reduced the WCET.

Superblock formation also makes it possible for *coding sinking* to reduce the WCET. *Coding sinking* moves instructions inside a block downward following the control-flow into the successor blocks. Since *superblock formation* eliminates the *join* transitions along the WC path, the instructions being pushed down can be merged with instructions in the successor block along the worst-case path. This would reduce the execution time along the worst-case path. If a block has two successors, the moved instructions have to be duplicated for each successor. So code sinking can potentially increase the code size. 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.

10.4 Experimental Results

This section describes the results of a set of experiments to illustrate the effectiveness of improving the WCET by using WCET path optimizations. All of the optimizations described in the previous sections were implemented in the compiler and the measurements were automatically obtained after applying these optimizations. Note that the *DSPStone* benchmarks are not used for the experiments on WC path optimizations for the same reason as *WCET code positioning*. The benchmark *findmax* contains the code similar to the example shown in Figure 10.4. In the example in Figure 10.4, 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 10.3(a). However, in the benchmark *findmax*, the initial value for i in the *for* loop is assigned 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 10.3(b).

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 transfer 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 in both experiments [6].

Table 10.1 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.

Table 10.1. WCET 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	7,491,544	0.984	1.009	1.40	7,245,535	0.951	1.557	2.40	7,241,047	0.951	1.557	5.47
findmax	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	0.949	1.018	1.43	18,128	0.929	1.821	2.57	16,325	0.836	1.804	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	0.718	1.051	1.63	16,547	0.718	1.410	2.38	16,546	0.718	1.013	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	1.089	1.38	321,017	0.915	1.633	2.13	300,920	0.858	1.684	5.88
small average	802,732	0.953	1.059	1.36	777,137	0.923	1.422	2.30	773,248	0.859	1.511	5.77
bitcnt	55,521	0.998	1.003	1.56	50,623	0.910	1.164	3.44	49,023	0.881	1.161	7.78
diskrep	12,492	1.000	1.000	1.75	12,492	1.000	1.000	3.38	11,905	0.953	1.021	17.44
fft	73,386	0.994	0.998	1.36	70,954	0.961	1.580	3.86	70,891	0.960	1.583	15.24
fire	9,679	0.948	1.105	1.67	9,539	0.934	1.765	4.00	9,395	0.920	1.789	15.33
sha	759,208	0.987	1.000	2.15	733,092	0.953	1.218	12.33	733,450	0.953	1.225	39.96
stringsearch	194,113	0.998	1.039	2.03	173,821	0.894	1.447	5.41	167,893	0.963	1.432	15.34
larger average	184,067	0.987	1.024	1.75	175,087	0.942	1.362	5.40	173,760	0.922	1.369	18.51
overall average	570,732	0.966	1.046	1.51	551,368	0.930	1.399	3.46	548,440	0.882	1.457	10.55

The results after *WC path duplication* shown in the middle portion of Table 10.1 were obtained by performing *superblock formation* followed by WC *path duplication.* If the WCET did not improve, the transformations are then discarded. 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 less transfer of control 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 10.1 were obtained by performing *superblock formation*, WC *path duplication*, and WCET *code positioning.*

Table 10.2. 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

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 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.

The effect on ACET after applying *superblock formation*, *path duplication*, and *code positioning* to improve WCET is shown in Table 10.2. 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*. It appears that WCET *code positioning* typically helped ACET in Table 9.4. 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 10.2 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 10.3 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 10.3) are depicted. Finally, the results after WCET code positioning are given. As expected, *loop unrolling* reduced WCET for all benchmarks. If typical input data was 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*. 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 10.3 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 10.1 and 10.3. 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 significant percentage of the total code size. Performing these optimizations on larger applications should result in a smaller percentage code size increase. 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 on the timing constraints and code size limitation that should be met.

The *time ratio* columns in Tables 10.1 and 10.3 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 compared to those cited in a previous study [6]. First, the applied optimizations increased the number of

Table 10.3. WCET Results after Loop Unrolling and Superblock Formation

Program	After Loop Unrolling				After Superblock Formation				After Code Positioning			
	WCET		Size	Time	WCET		Size	Time	WCET		Size	Time
	Cycles	Ratio	Ratio	Ratio	Cycles	Ratio	Ratio	Ratio	Cycles	Ratio	Ratio	Ratio
bubblesort	7,242,043	0.951	1.313	1.00	7,118,791	0.935	1.774	2.20	7,115,798	0.934	1.765	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	0.846	1.411	1.29	16,520	0.846	1.411	2.57	14,728	0.755	2.143	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	0.871	1.500	1.88	17,048	0.740	1.782	4.88	15,548	0.675	1.782	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	1.620	1.25	311,017	0.887	2.177	2.88	311,017	0.887	2.177	6.50
small average	777,375	0.904	1.478	1.63	762,080	0.858	1.761	3.51	760,931	0.823	1.857	9.23
bitcnt	50,720	0.912	1.113	1.06	49,220	0.885	1.121	2.61	47,720	0.858	1.113	5.22
diskrep	12,096	0.968	1.242	2.38	12,094	0.968	1.242	11.75	11,713	0.937	1.258	29.69
fft	73,440	0.995	1.203	1.07	72,234	0.978	1.192	2.00	72,178	0.978	1.197	5.60
fire	9,890	0.969	1.255	1.22	9,199	0.901	1.696	4.00	9,184	0.900	1.704	19.78
sha	733,532	0.953	1.092	1.06	712,499	0.926	1.086	4.08	712,467	0.926	1.093	14.42
stringsearch	186,019	0.956	1.330	1.25	184,594	0.949	1.417	4.19	179,578	0.923	1.441	14.19
larger average	177,616	0.959	1.206	1.34	173,307	0.935	1.293	4.77	172,140	0.920	1.301	14.81
overall average	552,465	0.925	1.376	1.52	541,290	0.887	1.586	3.99	540,134	0.860	1.649	11.33

basic blocks and paths in the program, which increased the time needed for timing analysis and required additional invocations of the timing analyzer for WCET code positioning. Second, required phases (*fixing the entry/exit* of the function was performed 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 these phases. These required transformations was discarded after invoking the timing analyzer 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. In contrast, WCET *code positioning* is performed after

Table 10.4. 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

these required phases. Thus, there is no need to discard and re-apply transformations after performing WCET *code positioning.*

The effect on ACET after WCET path optimization for the second experiment is shown in Table 10.4. For the benchmarks in Table 10.4, *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.

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 chapter 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 10.1 and 10.3, WCET code positioning has a much greater impact on compilation time.

10.5 Conclusions

This chapter describes how the WCET of a program can be reduced by optimizing the WC paths. Our compiler automatically uses feedback from the timing analyzer to detect the WCET paths through a function. There are two contributions in this chapter. First, traditional frequent path optimizations are applied to WC paths and improvements in the WCET are obtained. Second, new optimizations are developed, such as WC *path duplication* and *loop unrolling* for an odd number of iterations to improve WCET while minimizing code growth.

Several 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.

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 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, reasonable WCET improvements can still be achieved by optimizing the WC paths of an application.

CHAPTER 11

FUTURE WORK

There are many areas of future research that can be investigated for WCET reduction. These WCET optimizations described in this dissertation may be applied for different architectures and a larger class of applications. In addition, the compilation time can be improved and the tradeoff between WCET and code size could be evaluated.

These WCET compiler optimizations could be used to improve WCETs on other processors. The genetic algorithm searching for optimization phase sequences and the WC path optimizations may be applied for other processors. Besides improving WC performance by reducing the transfers of control along the WC paths, the WCET code positioning algorithm may be adapted to improve instruction cache performance along the WC path for processors with caches. In addition, WC path optimizations could be applied for processors that provide support instruction level parallelism.

The WCET compilation time can be further reduced. Longer compilation times may be acceptable for embedded systems since developers may be willing to wait longer for more efficient executables. However, people working in industry still want to reduce the compilation time so they have more time to try more options while developing software for embedded systems. Now the compiler and the timing analyzer are currently separate processes and they exchange data via files. If the compiler and the timing analyzer could be merged into one process, then it would speed up the compilation. In addition, the WCET optimization algorithms may also be improved to speed up the compilation.

Currently, the WCET optimizations are applied on benchmarks with bounded loops since the timing analyzer can give exact clock cycles for the WCET of these programs. If the timing analyzer can produce a WCET with a symbolic number of loop iterations as the parameters, then these optimizations can still be performed to reduce the WCET for programs whose number of iterations cannot be statically determined by the compiler [34].

The characteristics of the genetic algorithm search can be varied. It would be interesting to see the effect on the WCET as one changes aspects of the genetic algorithm search, such as the sequence length, population size, number of generations, etc.

Currently, the compiler performs WC path optimizations on all of 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 on 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. The compiler can be enhanced 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.

CHAPTER 12

CONCLUSIONS

This dissertation has presented a compilation system, which is integrated with a timing analyzer, to perform three different types of compiler optimizations to reduce WCET. First, this system allows a user to invoke a genetic algorithm that automatically searches for an effective optimization phase sequence for each function that best reduces the WCET. Second, WCET code positioning is used to reorder or position the basic blocks within a function to reduce WCET. Finally, path-based optimizations are applied on WC paths to reduce WCET.

This dissertation has made several contributions. To the best of our knowledge, this is the first compiler that interacts with a timing analyzer during the compilation of applications. It has been shown that WCET predictions can be used as a fitness criteria by a genetic algorithm that finds effective optimization sequences to improve the WCET of applications on an embedded processor. A WCET code positioning algorithm was developed, which is driven by WCET path information from timing analysis, as opposed to ACET frequency data from profiling. Experimental results show that the greedy WCET code positioning algorithm obtains optimal results on the SC100 for the suite of programs with a small number of basic blocks. In addition, it is the first time that path optimizations are applied on WC paths and improvements in the WCET are obtained. Finally, we have shown that these WC optimizations improve ACET on average.

This dissertation has presented the first general study on compiler optimization techniques to reduce WCET. An improvement in the WCET of a task may enable an embedded system to meet timing constraints that were previously infeasible. The WCET information from the timing analyzer is used to guide the compiler optimizations. One advantage of using WCET, instead of ACET from profiling, is that the timing analysis is much faster than simulation, which is required to obtain profile data on many embedded development environments. In addition, the process is entirely automatic, unlike profile driven optimizations, which requires the user to provide typical input data. Finally, the different compiler optimization techniques described in this dissertation have been shown to significantly reduce WCET.

REFERENCES

[1] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, M. Craven, "Energy Management for Real-Time Embedded Applications with Compiler Support" *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 284-292 (June 2003).

[2] J. Eyre, and J. Bier, "DSP Processors Hit the Mainsteam" *IEEE Computer*, 31(8) pp. 51-59 (August 1998).

[3] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones, 2002. "VISTA: A System for Interactive Code Improvement," *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 155-164 (June 2002).

[4] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley J. Davidson, M. Bailey, Y. Paek, K. Gallivan, 2003. "Finding Effective Optimization Phase Sequences" *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 12-23 (June 2003).

[5] 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.* IEEE Computer Society, Toronto, Canada, pp. 472–481(2004).

[6] W. Zhao, D. Whalley, C. Healy, and F. Mueller, "WCET Code Positioning," *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE Computer Society, Lisbon, Portugal, pp. 81–91(2004).

[7] W. Zhao, W. Kreahling, D. Whalley, C. Healy, F. Mueller, "Improving WCET by Optimizing Worst-Case Paths," *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE Computer Society, San Francisco, United States, pp. 138–147(2005).

[8] A.C. Shaw, "Reasoning About Time in Higher- level Language Software,"*IEEE Transactions on Software Engineering*,15(7) pp. 875–889 (July 1989).

[9] C.Y.Park and A.C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Proceedings of the 11th Real-Time Systems Symposium*, pp. 72–81 (1990).

[10] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim, "An Accurate Worst Case Timing Analysis Technique for

RISC Processors,"*Proceedings of the Fifteenth IEEE Real-time Systems Symposium*, pp. 875–889 (December 1994).

[11] M. Harmon, T. Baker, D. Whalley,1994 "A Retargetable Technique for Prediction Execution Time of Code Segments," *Real-time Systems*, pp. 159–182 (September 1994).

[12] R. Arnold, F. Mueller, D. Whalley,1994. "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the Fifteenth IEEE Real-time Systems Symposium*, pp. 172–181 (December 1994).

[13] C. Healy, D. Whalley, M. Harmon, 1995. "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the Sixteenth IEEE Real-time Systems Symposium*, pp. 288–297 (December 1995).

[14] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, 1996. "Supporting the Specification and Analysis of Timing Constraints" *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pp.170-178 (June 1996).

[15] F. Mueller, 1997 "Timing Predictions for Multi-Level Caches," *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-time Systems*, pp.29-36 (June 1997).

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

[17] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, 1999 "Timing Constraint Specification and Analysis" *Software Practice & Experience*, pp.77-98 (January 1999).

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

[19] C. Healy, D. Whalley,1999 "Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 79–99 (June 1999).

[20] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon,1999. "Timing Analysis for Data Caches and Set-Associative Caches," *Real-Time Systems*, pp.209-233 (November 1999).

[21] F. Mueller, 2000 "Timing Analysis for Instruction Caches," *Real-Time Systems*, 18(2) pp.209-233 (May 2000).

[22] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, R. van Engelen, 2000 "Supporting Timing Analysis by Automatic Bounding of Loop Iterations," *Journal of Real-Time Systems*, pp. 121–148, (May 2000).

[23] C. Healy, D. Whalley, R. van Engelen,2000 "A General Approach for Tight Timing Predictions of Non-Rectangular Loops," *WIP Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 11–14, (May 2000).

[24] C. Healy, D. Whalley, 2000 "Automatic Detection and Exploitation of Branch Constraints for Timing Analysis," *IEEE Transaction on Software Engineering*, 28(8) pp. 763–781 (August 2000).

[25] Y. 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).

[26] J. Engblom and A. Ermedahl, "Modeling Complex Flows for Worst-Case Execution Time Analysis," *Proceedings of the 21st IEEE Real-time System Symposium*, pp. 875–889 (December 2000).

[27] T. Lundqvist and P. Stenstrom, "Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques,"*Proceedings of SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems(LCTES'98)*, pp. 1–15 (June 1998).

[28] T. Marlowe, S. Masticola, 1992. "Safe Optimization for Hard Real-Time Programming," *Special Session on Real-Time Programming, Second International Conference on Systems Integration*, pp. 438–446 (June 1992).

[29] S. Hong and R. Gerber, 1993. "Compiling Real-Time Programs into Schedulable Code," *Proceedings of the SIGPLAN'93*, pp. 166–176 (June 1993).

[30] 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).

[31] Star Core, Inc., and Atlanta, GA, SC110 DSP Core Reference Manual, 2001

[32] D. Hanson, C. Fraser "A Retargetable C Compiler," *Addison-Wesley Publishing Company* 1995.

[33] Star Core, Inc., and Atlanta, GA, SC100 Simulator Reference Manual, 2001

[34] E. Vivancos, C. Healy, F. Mueller, and D. Whalley, "Parametric Timing Analysis" *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pp. 88-93 (June 2001).

[35] K, Andrews, R. Henry, and W. Yamamoto, "Design and implementation of the uw illustrated compiler," *ACM SIGPLAN Conference on Programming Language Design and Implementation.* pp.105-114 (1988).

[36] M. Boyd and D. Whalley, "Isolation and analysis of optimization errors," *ACM SIGPLAN Conference on Programming Language Design and Implementation.* pp.26–35 (1993).

[37] M. Boyd and D. Whalley "Graphical visualization of compiler optimization," *Programming Languages 3*, pp. 69–94(1995).

[38] M. Benitez and J. Davidson "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN'88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[39] M. Benitez "Retargetable Register Allocation", *PhD Dissertation, University of Virginia, Charlottesville, VA.* 1994

[40] M. Benitez and J. Davidson "The Advantages of Machine-Dependent Global Optimization" *Proceedings of the 1994 International Conference on Programming Languages and Architectures, in Zurich, Switzerland.* pp. 105-124 (1994).

[41] S. C. Johnson "Yacc - Yet Another Compiler Compiler," *Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.* 1975.

[42] B. Appelbe, K. Smith, and C. McDowell, "Start/pat: a parallel- programming toolkit", *IEEE Software*, 4, vol. 6. 29 (1989).

[43] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors", *International Journal of High Speed Computing.* 1, vol. 1. Pensylvania State University Press, pp. 39-48 (1989).

[44] J. Browne, K. Sridharan, J. Kiall, C. Denton, and W. Eventoff, "Parallel structuring of real-time simulation programs", *COMPCON Spring '90: Thirty-Fifth IEEE Computer Society International Conference*, pp. 580-584(1990).

[45] C. Dow, K. Chang, and M. Soffa, 1992. "A visualization system for parallelizing programs", *Supercomputing.* pp. 194-203(1992).

[46] S. Liao, A. Diwan, R. Bosch, A. Ghuloum, and M. Lam, 1999. "Suif explorer: an interactive and interprocedural parallelizer", *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM Press, pp. 37-48(1999).

[47] S. Novack and A. Nicolau, "Vista: The visual interface for scheduling transformations and analysis" *Languages and Compilers for Parallel Computing*, pp. 449-460(1993).

[48] B. Harvey, and G. Tyson, "Graphical user interface for compiler optimizations with simple-suif", *Technical Report UCR-CS-96-5, Department of Computer Science, University of California Riverside, Riverside, CA.* (1996).

[49] H. Massalin, "Superoptimizer: a look at the smallest program" *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating systems.* pp. 122-126(1987).

[50] T. Granlund, and R. Kenner, "Eliminating branches using a superoptimizer and the gnu c compiler", *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 341–352(1992).

[51] K. Chow, and Y. Wu, "Feedback-directed selection and characterization of compiler optimizations," *Workshop on Feedback-Directed Optimization.* 1999.

[52] T. Kisuki, P. Knijnenburg, and M. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation", *IEEE PACT.* pp 237-248(2000).

[53] R. Whaley, A. Petitet, and J. Dongarra, " Automated empirical optimization of software and the atlas project" *Parallel Computing.* 1-2, vol. 27. pp. 3-25(2001).

[54] P. Knijnenburg, T. Kisuki, K. Gallivan, and M. O'Boyle, "The effect of cache models on iterative compilation for combined tiling and unrolling", *Proc. FDDO-3.* pp 31-40(2000).

[55] D. Gannon, J. Lee, B. Shei, S. Sarukkai, S. Narayana, N. Sundaresan, D. Attapatu, and F. Bodin, " SIGMA II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems" *Proceedings of the IFIP Edinburgh Workshop on Parallel Programming Environment*, April 1992.

[56] F. Bodin, E. Rohou, and A. Seznec, " SALTO: System for Assembly-Language Transformation and Optimization" *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, December 1996.

[57] A. Nisbet, "Genetic algorithm optimized parallelization", *Workshop on Profile and Feedback Directed Compilation.* 1998.

[58] K. Cooper, P. Schielke, and D. Subramanian "Optimizing for Reduced Code Space using Genetic Algorithm," *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp.1-9 (May 1999).

[59] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. "Finding Effective Compilation Sequences," *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 231-239 (June 2004).

[60] F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Proceedings of the IEEE Real-Time Systems Symposium*, pp.179-188 (June 1999).

[61] P. Puschner and R. Nossa, "Testing the Results of Static Worst-Case Execution Time Analysis," *Proceedings of the IEEE Real-Time Systems Symposium*, pp.134-143 (December 1999).

[62] F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Real-Time Systems*, 21(3) pp.241-268 (November 2001).

[63] J. Holland, "Adaptation in Natural and Artificial Systems", Addison-Wesley (1989)

[64] K. Pettis and R. Hansen "Profile Guided Code Position," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 16–27 (June 1990).

[65] S. McFarling and J. Hennessy "Reducing the Cost of Branches," *13th Annual International Symposium of Computer Architecture*, pp. 396–403 ACM, (1986).

[66] B. Calder and D. Grunwald "Reducing Branch Costs via Branch Alignment," *Proceeding of ASPLOS'94*, pp. 242–251 (October 1994).

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

[68] W. Hwu, S. Mahlke, W. Chen, P. Change, 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).

[69] F. Mueller and D. Whalley "Avoiding Unconditional Jumps by Code Replication," *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation* pp. 322–330 (June 1992).

[70] F. Mueller and D. Whalley "Avoiding Conditional Branches by Code Replication," *Proceedings of the SIGPLAN '95 Conference on Programming Languages Design and Implementation* pp. 56–66 (June 1995).

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

BIOGRAPHICAL SKETCH

Wankang Zhao

The author was born on July 18, 1967, and received his PhD in Mechanical Manufacturing from Tianjin University in 1993. He was an associate professor in Nanjin University of Post and Telecommunications. At Florida State University, he received his M.S. in Computer Science in 2001 and his Ph.D in Computer Science in 2005.