Improving WCET by Applying a WC Code-Positioning Optimization

WANKANG ZHAO and DAVID WHALLEY
Florida State University
CHRISTOPHER HEALY
Furman University
and
FRANK MUELLER
North Carolina State University

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 transformation, which reduce the WCET, since an application with a lower WCET will be less likely to violate its timing constraints. Some processors incur a pipeline delay whenever an instruction transfers control to a target that is not the next sequential instruction. Code-positioning optimizations attempt to reduce these delays by positioning the basic blocks to minimize the number of unconditional jumps and taken conditional branches that occur. Traditional code-positioning algorithms use profile data to find the frequently executed edges between basic blocks, then minimize the transfers of control along these edges to reduce the average case execution time (ACET). This paper introduces a WCET code-positioning optimization, driven by the worst-case (WC) path information from a timing analyzer, to reduce the WCET instead of ACET. This WCET optimization changes the layout of the code in memory to reduce the branch penalties along the WC paths. Unlike the frequency of edges in traditional profile-driven code positioning, the WC path may change after code-positioning decisions are made. Thus, WCET code positioning is inherently more challenging than ACET code positioning. The experimental results show that this optimization typically finds the optimal layout of the basic blocks with the minimal WCET. The results show over a 7% reduction in WCET is achieved after code positioning is performed.

Categories and Subject Descriptors: B.3.2 [**Memory Structure**]: Design Style-Pipelining; C.1.0 [**Processor Architecture**]: General; C.4 [**Performance of Systems**]: Measurement Techniques-WCET; D.3.4 [**Programming Languages**]: Processors-Compilers, optimization

A preliminary version of this paper entitled "WCET Code Positioning" W. Zhao, D. Whalley, and F. Mueller, appeared in the Proceedings of the *IEEE Real-Time Systems Symposium* (Dec.), 2004. 81–91.

Authors' addresses: W. Zhao and D. Whalley, Computer Science Department, Florida State University, Tallahassee, Fl 32306; C. Healy, Computer Science Department, Furman University, Greenville SC 29613; F. Mueller, Computer Science Department, North Carolina State University, Raleigh, NC 27607.

General Terms: Algorithms, Measurement, Performance
Additional Key Words and Phrases: WCET, embedded systems, code positioning

1. INTRODUCTION

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

The *worst-case execution time* (WCET) 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 accurately predict the WCET, but to also improve it. An improvement in the WCET of a task may enable an embedded system to meet timing constraints that were previously infeasible. WCET constraints can impact power consumption as well. In order to conserve power, one can determine the WC number of cycles required for a task and lower the clock rate to still meet the timing constraint with less slack. Improving the WCET of a task may allow an embedded system developer to use an even lower clock rate and save additional power, which is valuable for mobile applications. In contrast, conservative assumptions concerning WCET may require the processor to be run at a higher clock rate, which consumes more power.

Many DSP benchmarks represent kernels of common applications where most of the cycles occur. Such kernels in DSP applications have been historically written and optimized by hand in assembly code to ensure high performance [Eyre and Bier 1998]. However, assembly code is less portable and is harder to develop, debug, and maintain. Many DSP 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 manually written in assembly, aggressive compiler optimizations are used to ensure high performance.

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 are omitted in order to reduce the complexity of

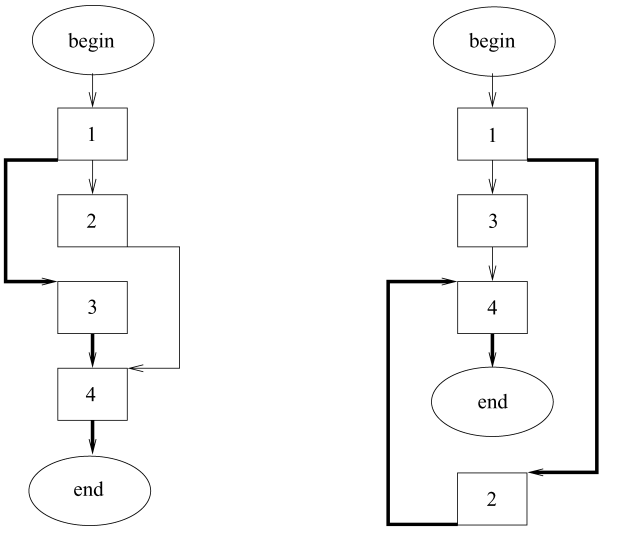

Fig. 1. Code positioning for an *if-then-else* statement.

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. The goal of traditional code positioning is to improve the *average-case execution time* (ACET), the typical execution time for a program.

In contrast, in this paper we describe an approach for improving 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. 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. A typical CFG for an *if-then-else* statement is shown in Figure 1 with two paths, $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$. Before *code positioning* (Figure 1a), assume that path $1 \rightarrow 3 \rightarrow 4$ is the WC path. After blocks 3 and 4 are physically moved in memory next to block 1 to remove the branch penalty from block 1 to block 3 (Figure 1b). path $1 \rightarrow 2 \rightarrow 4$ may become the WC 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 contrast, the frequency of the

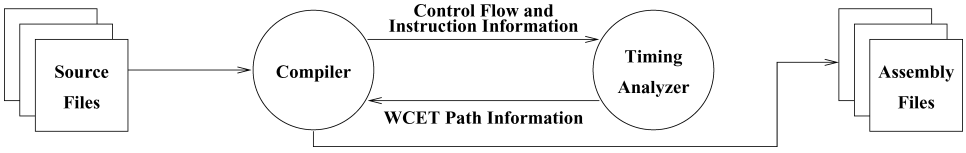

Fig. 2. The compiler interacts with the timing analyzer to obtain the WC path information.

edges based on profile data, which is used in traditional code positioning, does not change, regardless of how the basic blocks are ordered. Thus, WCET code positioning is inherently more challenging than ACET code positioning.

We use the following approach for improving the WCET of an application by code positioning. The WC path may change after code-positioning decisions are made. Therefore, we integrate a timing analyzer with a compiler where the WCET path information of both the application and the current function can be calculated on demand. Thus, the timing analyzer is invoked to determine the up-to-date WCET path information through a function, each time an edge in the CFG is selected for positioning. Calculating WCET using a timing analyzer places restrictions, such as requiring all loops be structured and bounded, on the kinds of programs that our compiler can automatically optimize. We describe the full set of such restrictions in Section 4. Figure 2 shows how the compiler obtains the WCET after each positioning step. The compiler sends information about the control flow and the current instructions that have been generated to the timing analyzer. Predictions regarding the WCET are sent back to the compiler from the timing analyzer. After positioning all of the edges of the CFG in a function, we also align the blocks, which are targets of transfers of control to further reduce WCET by minimizing target misalignment penalties. We retargeted both the VPO (*very portable optimizer*) compiler [Benitez and Davidson 1988; Benitez 1994; Benitez and Davidson 1994] and our timing analyzer to the StarCore SC100 processor for this study.

The major contributions that are presented in this paper are:

1. The first implementation of a WCET code-positioning optimization. All other code-positioning optimizations are designed to improve ACET, instead of WCET. In fact, we show in this paper that our WCET code-positioning algorithm obtains optimal results for all of our small test programs.
2. This is the first compiler optimization, of which we are aware, that is designed and implemented to improve WCET as opposed to ACET. WCET code positioning is driven by WCET path information obtained from a timing analyzer, as opposed to traditional code positioning, which uses frequency data obtained from profiling. Thus, we have demonstrated that timing analysis can be used to not only measure the WCET, but also as input to a compiler to improve the WCET.

The remainder of the paper is organized as follows. In Section 2, we outline the work related to this research, including WCET predictions, WCET reduction techniques, and traditional code-positioning algorithms. The underlying architecture used for this research is the StarCore SC100. In Sections 3 and 4,

we introduce the StarCore SC100 processor and the retargeting of our timing analyzer.

In Section 5, we present how our timing analyzer was integrated with our compiler to obtain the WC path information. We discuss in Section 6 the code-positioning algorithm to improve the WCET layout of the code in memory. In Section 7, we describe a related optimization to align blocks that are targets of transfer of control. We give the experimental results in Section 8. In Section 9 we describe future work in this research area. Section 10 presents conclusions.

2. RELATED WORK

This section summarizes the prior research on WCET predictions, WCET reduction techniques, and code-positioning optimizations.

2.1 Prior Work on WCET Predictions

There has been much work on WCET predictions. Each of the general techniques for predicting WCET is described in the following subsections. None of these techniques relies on requiring the user to specify restrictions on the input data values. 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.

2.1.1 *Timing Schema.* In the 1980s, Shaw started using timing schema to predict the maximum execution time of real-time programs [Shaw 1989]. A WCET prediction tool has been developed based on a timing schema, which is associated with each source-level program language construct [Lim et al. 1994]. 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 and machine codes. This limitation prevents the analysis of optimized code. Thus, the WCET from the timing schema cannot easily be 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 [Harmon et al. 1994; Arnold et al. 1964; Healy et al. 1995, 1999; 2000a, b; Ko et al. 1996, 1999; Mueller 1997, 2000; Healy and Whalley 1999, 2000; White et al. 1997, 1999]. 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 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 [Ko et al. 1999]. 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, such as WCET code positioning.

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 [Li et al. 1995; Engblom and Ermedahl 2000]. 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 [Lundqvist and Stenstrom 1998]. 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. However, 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 timing constraints of critical portions in a task [T. Marlowe 1992]. 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 [1993] 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. 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 portions 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. [2004] used WCET information to choose how to generate code on a dual-instruction set processor for the ARM and the Thumb. ARM code is generated for a selected subset of basic blocks that can impact the WCET. Thumb code is generated for the remaining blocks to minimize code size. In this way, they can reduce WCET while minimizing code size. Thus, they are using WCET information to choose the instruction set to select, when generating code. In contrast, we attempt to improve the WCET on a single-instruction set processor.

2.3 Prior Work on Code Positioning

Several code positioning (basic block reordering) approaches have been developed. Pettis and Hansen [1990] used execution profile data to position the code in memory. Profile data is 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 are 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 [1986] described a number of code-positioning methods to reduce branch misprediction and instruction fetch penalties. Calder and Grunwald [1994] proposed an improved code-positioning algorithm using a model to evaluate the cost of different basic block orderings. They assign 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, we describe a code-positioning algorithm in this paper to improve the WCET based on the WCET path information from the timing analyzer. WCET code positioning is inherently more complicated than ACET code positioning, since the WC path may change after positioning code in memory.

3. 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 has 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 [Star Core 2001b]. 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. There are 16 data registers and 16 address registers in the register file. Data registers are used to store data and address registers are used to store addresses. Each data register is 40 bits long and each address register is 32 bits long. The size of instructions can vary from one word (two bytes) to five words (ten bytes) depending upon the type of instruction, addressing modes used, and register numbers that are referenced. The SC100 machine has DSP addressing modes, which are register indirect (Rn), postincrement (Rn)+, postdecrement (Rn)−, postincrement by offset (Rn)+Ni, index by offset (Rn+N0), indexed (Rn+Rm), and displacement (Rn+x).

The SC100 has a five-stage pipeline machine. 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 no branch prediction or target buffers 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 there is a delay of three cycles. If it is not taken, then no delay is incurred. Unconditional jump instructions take two extra cycles if they use absolute addresses 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 fetches instructions in sets of four words that are aligned on eight-byte boundaries. The target of a transfer of control is

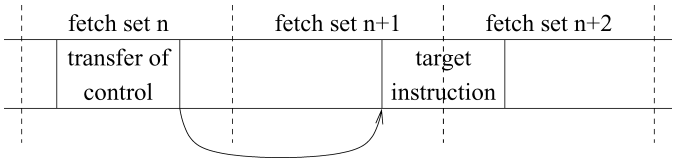

Fig. 3. Example of a misaligned target instruction.

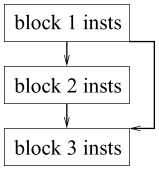

Fig. 4. Example: Branch penalty affects the worst-case path.

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. In this situation, the processor stalls for an additional cycle after the transfer of control [Star Core 2001b].

The transfer of control and branch target-alignment penalties for the SC100 can lead to nonintuitive WCET results. For instance, consider the flow graph in Figure 4. 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. Measurements indicating the accuracy of the WCET predictions produced by our timing analyzer will be shown later. In general, we could produce fairly accurate WCET predictions since some of the more problematic issues, which include a memory hierarchy and an operating system, are not present on this processor.

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 [Star Core 2001b].

4. SC100 TIMING ANALYZER

In this section we describe the timing analyzer that is used in this study [Harmon et al. 1994; Arnold et al. 1994; Healy et al. 1995, 1999; 2000a, b; Ko et al. 1996, 1999; Mueller 1997, 2000; White et al. 1997, 1999; Healy and Whalley 1999, 2000] and how we retargeted it to the SC100 machine. Like other timing analyzers, we enforce several restrictions to an application in order to accurately predict its WCET. The call graph must be explicit, so no calls through pointers are allowed. All loops must be bounded where the maximum number

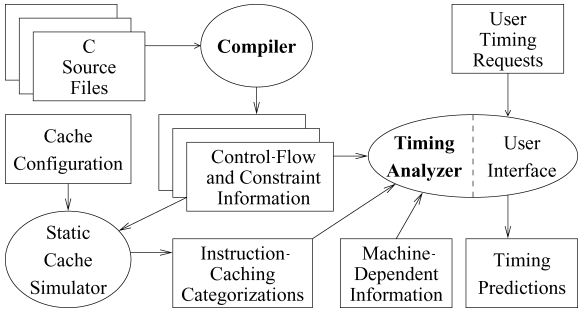

Fig. 5. An overview of the existing process to obtain WCET predictions.

of iterations can be statically determined by the compiler. Furthermore, all loops or cycles of code in the application must be recognized. Thus, we do not allow unstructured loops or recursion. Finally, we do not allow system calls. For the SC100 this means that no floating-point or integer divide operations are allowed, since there is no direct support for these operations in the SC100 processor.

Figure 5 depicts the organization of the framework that was used in the past by authors 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 issues:

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

An instruction's execution time can vary greatly depending on whether that instruction causes a cache hit or miss. The timing analyzer starts by performing WC cache analysis [Arnold et al. 1994]. 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 [Healy et al. 1995]. 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 [Healy et al. 1999]. One type of constraint is determining the maximum number of iterations

associated with each loop, including nonrectangular loops, where the number of iterations of an inner loop depends on the value of an outer loop variable [Healy and Whalley 1999]. Another type of constraint is branch constraints [Healy and Whalley 2000]. The timing analyzer uses these constraints to detect infeasible paths through the code and the frequency that 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. 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 of the loop [Ko et al. 1999].

We retargeted the timing analyzer to the SC100 processor to demonstrate that we could predict the WCET for an embedded machine. It is difficult to produce cycle-accurate simulations for a general-purpose processor because of 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 has neither a memory hierarchy (no caches or virtual memory system) nor an OS. 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.

The pipeline information of the instructions in the timing analyzer had to be updated for the SC100 processor. There are several modifications to support timing analysis of applications compiled for the SC100. First, the machine-dependent information (see Figure 5) 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. Thus, the static cache simulation step shown in Figure 5 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 are not assessed this penalty. When there is a transfer-of-control penalty, the timing analyzer calculates the number of clock cycles of the delay, which depends on

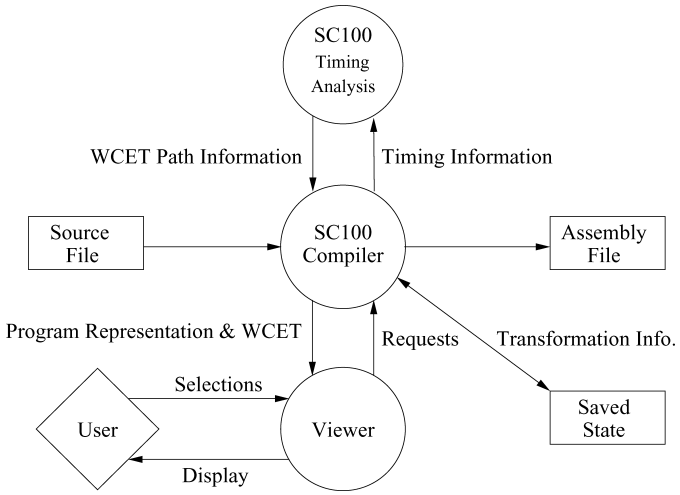

Fig. 6. An overview of the research framework.

the instruction type and whether there is a one-extra-cycle target misalignment penalty. Therefore, the size of each instruction is needed to detect when the target misalignment penalty will occur.

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

5. INTEGRATING THE TIMING ANALYZER WITH THE COMPILER

We use an interactive compilation system called VISTA (VPO Interactive System for Tuning Applications) [Zhao et al. 2002; Kulkarni et al. 2003] to experiment with our WCET code-positioning optimization. We retargeted VISTA to the SC100 architecture and integrated it with the SC100 timing analyzer, which provides the WCET path information to the compiler to perform the code-positioning optimization. Figure 6 shows an overview of the infrastructure used for this research. Note that Figure 5 only shows a portion of the framework shown in Figure 6, namely the timing analysis and compiler components. The SC100 compiler takes the source code and generates the assembly and the timing information, which is fed into the timing analyzer to obtain the WC path information. The compiler performs the code-positioning optimization and the user can see transformations to the program at the machine code level graphically. At each step of the code positioning, the compiler automatically invokes the timing analyzer to update the WC path information, which is used to guide the next step. The graphical user interface in the viewer helped

us to debug the problems when we implemented our WCET code-positioning algorithm.

The timing analyzer needs information as input to predict the WCET. This information is generated by the compiler as a side effect of producing the assembly. Each instruction in the assembly has a counterpart in the timing information where the instruction type and the registers set and used are presented. Besides the instruction data, this information contains the control-flow data, such as branches and loops, for the program. Many modifications to the SC100 VPO compiler were required to produce this information for timing analysis. For instance, the SC100 information contains the size for each instruction, which is needed by the timing analyzer to detect the branch target-misalignment penalties.

In order to measure the WCET, VPO generates the timing information for the benchmark, invokes the timing analyzer, saves the output of the timing analyzer to get the WCET and the path information, which is used to guide the code-positioning. The output 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 blocks in each path

Our timing analyzer calculates the WCET for all paths within each loop and the outer level of a function. A loop path consists of basic blocks and each loop path starts with the entry block (header) in the loop and is terminated by a block that has a transition back to the entry block (back edge) or transition outside the loop. A function path starts with the entry block to the function and is terminated by a block containing a return. If a path enters a nested loop, then the entire nested loop is considered a single node along that path. The WCET for each path is calculated by the timing analyzer. This WC path information is used by the compiler to position the blocks in each function.

The WCET for the whole program is calculated bottom-up based on a timing-analysis tree, where each node is a loop or a function (ref. Section 2.1.2). This code-positioning algorithm is also performed bottom up for each node in the timing-analysis tree. The timing analyzer provides the WCET and the lists of the basic blocks along all paths. Therefore, the compiler can perform code positioning one edge at a time to minimize the transfer-of-control penalties along the WC paths.

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 process since all required transformations have been performed on the code. After code positioning, the branch target-misalignment optimization is performed.

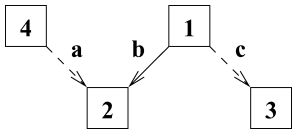

Fig. 7. Selecting an edge to be contiguous.

6. WCET CODE POSITIONING

In this section, we discuss the WCET code-positioning algorithm and present an example to illustrate the algorithm. Traditional code-positioning algorithms reduce the ACET by contiguously positioning the basic blocks within the frequent flow of control in memory. This code positioning might increase the execution time along the infrequent paths. However, it does not often adversely affect the ACET, since infrequent paths do not contribute as much to the overall execution time.

WCET code positioning needs to be driven by WCET path information instead of just the frequencies of the transitions between blocks obtained by profile data. Unfortunately the WC path may change each time the basic blocks are reordered. Increasing the execution time along any path may make that path the new WC path. The goal of WCET code positioning is to minimize the maximum execution time along all the paths. Thus, WCET code positioning is inherently a more complicated problem than ACET code positioning.

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

Our 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. However, not all edges can be contiguous. Consider the portion of a control-flow graph shown in Figure 7. 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.

If the timing analyzer calculates the WCET path information on the original positioned code, then changing the order of the basic blocks may result in unanticipated increases in the WCET for other paths, since previously contiguous edges may become noncontiguous. We decided instead to treat the basic blocks as being initially unpositioned so that the location of a basic block does not affect the WCET of the paths containing it. Thus, we actually modify the code so that all transitions between blocks are accomplished using a transfer of control

and will result in a transfer of control penalty. This means an unconditional jump is added after each basic block that does not already end with an unconditional transfer of control. Note that when WCET code positioning is applied, the unnecessary unconditional jumps are deleted.

The basic idea of our WCET code-positioning algorithm is to find the edge in the CFG that contributes the most to the WCET, which we call the WC edge, and make the two basic blocks linked by that edge contiguous to reduce the execution time along the WC path. This operation may result in a new WC path, so the algorithm positions one edge at a time and recalculates 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 WC edge among all the edges along the WC paths. Eliminating the transition penalty at the chosen edge will reduce the execution time along the WC 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 our WCET code-positioning algorithm can be presented. Edges are denoted as being contiguous, noncontiguous, or unpositioned. A *contiguous* edge has its source block immediately positioned before its target block in memory. In contrast, a *noncontiguous* edge does not. An *unpositioned* edge means that it has not yet been determined if it will be contiguous or noncontiguous. The *upper bound* WCET (UB-WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be noncontiguous. In other words, the inserted unconditional jumps are included in the WCET of a path. The *lower bound* WCET (LB-WCET) of a path indicates the WCET when all current unpositioned edges are assumed to be contiguous. Thus, the inserted unconditional jumps are not included in the WCET of a path. Note that the deletion of unconditional jumps may result in additional no-ops being inserted for the SC100, since these hazards may have previously been overlapped with the execution of an unconditional jump. The *weighted* WCET for a path is the WCET for a single iteration of a path times 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 8. Note that the algorithm is implemented in the compiler, which gets the WCET path information from the timing analyzer, as shown in Figure 6.

At this point target-misalignment penalties are not assessed by the timing analyzer, since WCET target alignment, described in Section 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 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

```
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
      first if it is contributing, second its weighted Upper-Bound
      WCET (UB_WCET), and third its weighted Lower-Bound WCET (LB_WCET).

   /* choose the best_edge in the path that is also used in the
      next highest ranked path */
   max = -1;
   FOR (each unpositioned edge e in path p ) {
      n = 0;
      FOR (each path p in the sorted path list (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;

}
```

Fig. 8. The pseudocode for the WCET code-positioning algorithm.

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. This greedy algorithm attempts to choose the *worst-case* edge at each point in the positioning.

Consider the source code in Figure 9, which is a contrived example to illustrate the algorithm. Figure 10 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 10 have conditions that are reversed from the source code to depict the branch conditions that are represented at the assembly instruction level.

```
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;
   }
}
```

Fig. 9. An example used to illustrate the algorithm.

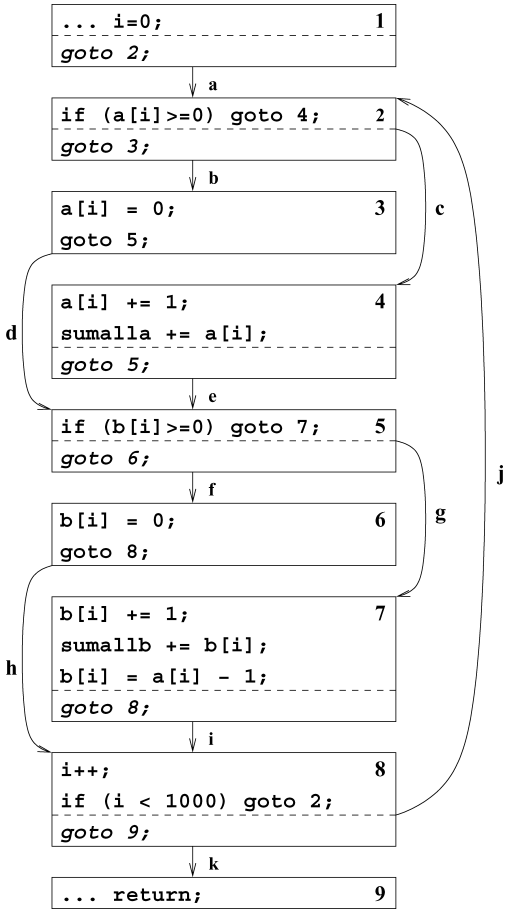

Fig. 10. Control-flow graph of code of an example.

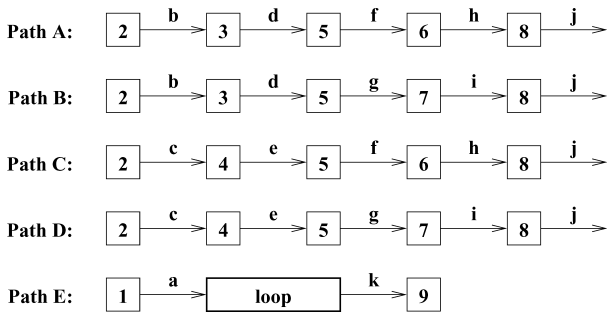

Fig. 11. Paths of the example in Figure 10.

Table I. Benchmarks Used in the Experiments[a]

Stp	Status of Edges Shown in Figure 10											WCETs of Paths Shown in Figure 11									
												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

[a] u, unpositioned; c, contiguous; n, noncontiguous.

Several unconditional jumps, represented in Figure 10 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 10. The transitions (directed edges) between nodes are labeled so they can be referenced later. Figure 11 shows the paths through the control-flow graph. Paths A–D represent paths within the loop. Path E represents the outer level path, where the loop is considered a single node within that path. We consider backedges (directed edges back to the entry point of the loop) to be part of the paths within the loop, since these edges can be traversed on all loop iterations, except for the last one. Likewise, the exit edges (directed edges leaving the loop) are considered part of the outer paths containing the loop, since an exit edge is executed, at most, once each time the loop is entered.

Table I shows how WCET code positioning is accomplished for the example shown in Figures 10 and 11. At each step the status for each edge and the current UB-WCET and LB-WCET for each path, calculated from the timing analyzer, are shown. Initially all edges are unpositioned, as shown in step 0. For each step, an edge is selected to be *contiguous* and one or more edges become *noncontiguous*. Thus, after each step, one or more paths have their UB-WCET reduced and one or more paths have their LB-WCET increased.

- In the first step, the algorithm selects edge j to be *contiguous*, since it reduces the UB-WCET of all four paths in the loop. This selection also causes edges a and k to become *noncontiguous*, which results in only a small increase for

the LB-WCET of the entire function (path E), since these edges are outside the loop.

- In the second step, edge i is selected, since it is part of path D, which contains the greatest current UB-WCET. The algorithm chooses edge i instead of another edge in path D, since edge i is also part of path B, which contains the second greatest WCET at that point. Note that edge g could have also been chosen at this point. Since path B and path D share edge i, the UB-WCET of the two paths decreases by three 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 increases by three 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 path B and path D decreases because edge g is a part of path B and Path D. The LB-WCET of path A and path C increases because edge f becomes *noncontiguous*, while path A and path 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. 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 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 10, 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 I, 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 $2 \rightarrow 3$. Connecting these edges by their common nodes, we are able to determine that six of the nine blocks should be positioned in the order $4 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 2 \rightarrow 3$. The remaining blocks, which are 1, 6, and 9, can be placed either before or after this contiguous set of blocks. In general, there may be several contiguous sets of blocks in a function and these sets can be placed in an arbitrary order. We always designate the entry block of the function as the first block in the final positioning to simplify the generation of the assembly code by our compiler and the processing by our timing analyzer. Note that the entry block can never be the target of an edge in the control flow, because of the prologue code for the function being generated in this block. The process of the code positioning for this example is summarized in Figure 12, where the contiguous edges have thicker transitions and the steps are identified in which the edges are positioned.

Figure 13 shows the final positioning of the code after applying the WCET code-positioning algorithm. By contrasting the code in Figure 10 with the final positioning in Figure 13, one can observe that performing the final positioning sometimes requires reversing branch conditions, changing target labels of

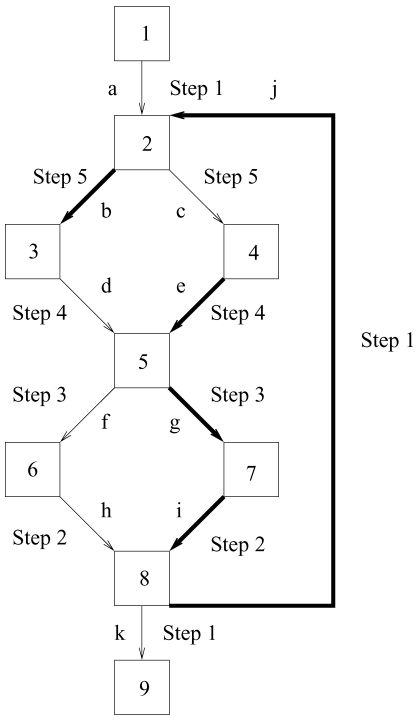

Fig. 12. Steps in which edges in Figure 10 are positioned.

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; paths B and D each require only one. Note that paths B and D had higher UB-WCETs before the edges were positioned.

The portion of our greedy algorithm (shown in Figure 8) that most affects the analysis time is the computation performed by the timing analyzer, which is invoked each time an edge is selected to become contiguous. Given that there are n basic blocks in a function, there can be, at most, n-1 contiguous edges; sometimes there are fewer. For instance, only five edges were selected to be contiguous instead of n-1 or eight edges for the example shown in Table I and Figure 13. 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.

7. 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, as illustrated in Figure 3. We attempt to add no-ops before the target instruction to make the target instruction fit into one fetch set. Figure 14 shows an example where the target instruction is in a single fetch

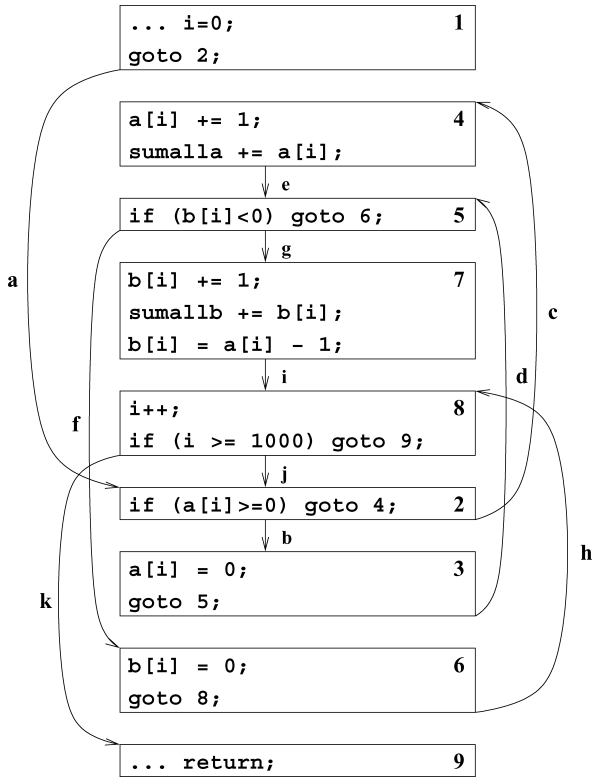

Fig. 13. Control flow graph of code of the example in Figure 10 after WCET positioning.

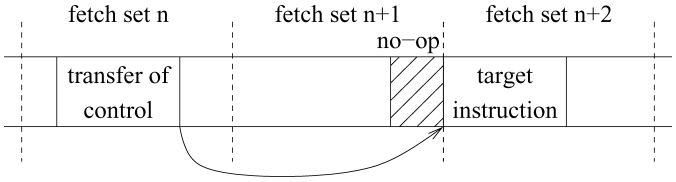

Fig. 14. Aligned target instruction.

set after we add a no-op instruction to force this misaligned target instruction to not span the fetch set boundary.

WCET target alignment attempts to minimize the number of target misalignment penalties in the following manner. 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 13 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 10 has three relocatable sets, which are {1, 2, 3}, {4, 5, 6}, and {7, 8, 9}.

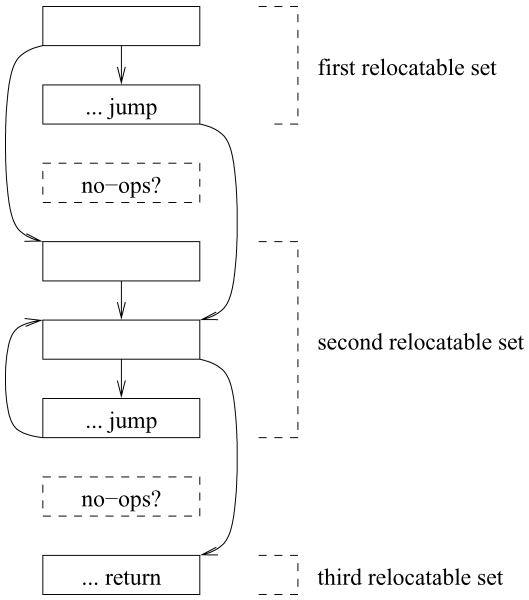

Fig. 15. Example of inserting no-op instructions before each relocatable set of blocks.

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 (two bytes) and each fetch set consists of four 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 is one word in size. The locations where no-ops can be inserted before each relocatable set of blocks is illustrated in Figure 15. 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 $4(m\text{-}1)$ times for each function, where m is the number of relocatable sets of blocks to be aligned. The 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, we added an option 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.

We could attempt a more aggressive approach 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, we

Table II. Benchmarks Used in the Experiments

	Program	Description
Small	bubblesort	performs a bubble sort on 500 elements
	findmax	finds the maximum element in a 1000-element array
	keysearch	performs a linear search involving four 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 50×50 matrix is symmetrical
	unweight	converts an adjacency 100×100 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

have found that the code size increase is small and our current approach is quite efficient.

8. RESULTS

This section describes the results of a set of experiments to illustrate the accuracy of the SC100 timing analyzer and the effectiveness of improving the WCET by using WCET code positioning and target alignment. Table II shows the benchmarks and applications used to test the effectiveness of these optimizations. These include benchmarks or programs used in previous studies by various groups (FSU, SNU, Uppsala) working on WCET timing analysis. These benchmarks in Table II were selected since they do have conditional constructs, which means the WCET and ACET input data may not be the same.

Traditionally, WCET benchmarks are quite small to make them amenable to timing analysis. First, it is often difficult to manually determine the WC input data to verify the accuracy of the timing analysis. Second, most timing analyzers, including ours, cannot make WCET predictions for programs containing unbounded loops, recursion, indirect calls, or calls to perform dynamic allocation or I/O. Thus, standard benchmark suites, such as *SPEC* or *MiBench*, are not used in timing analysis.

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 we modified the benchmark so that a global array is initialized with constants. Likewise, output is written to a global array.

Table III. Code Size and the Compilation Time of the Benchmarks

Category	Benchmarks	Code Size (bytes)	Lines of Source	Compilation Time		Time Ratio
				Base (min)	Position (min)	
Small	bubblesort	145	93	0.23	0.37	1.609
	findmax	58	21	0.97	0.97	1.000
	keysearch	186	537	0.18	0.22	1.222
	summidall	56	23	0.13	0.13	1.000
	summinmax	60	47	0.13	0.17	1.308
	sumnegpos	45	20	0.13	0.13	1.000
	sumoddeven	78	51	0.15	0.17	1.133
	sumposclr	81	35	0.15	0.20	1.333
	sym	97	40	0.18	0.18	1.000
	unweight	79	23	0.13	0.13	1.000
Small average		89	89	0.24	0.27	1.161
Larger	bitcnt	354	170	0.32	0.37	1.156
	diskrep	388	500	0.22	0.50	2.273
	fft	631	220	0.28	0.70	2.500
	fire	247	109	0.17	0.22	1.294
	sha	907	253	0.42	0.87	2.071
	stringsearch	333	237	0.40	0.97	1.425
Larger average		477	248	0.30	0.61	1.953
Overall average		234	149	0.26	0.39	1.458

In order to verify the accuracy of the WC timing analyzer, the SC100 simulator from StarCore is used to obtain the execution time driven by the WC input data. The WC input data is meticulously determined by hand through testing since the WC paths are often difficult to detect manually, because of the transfer of control penalties. Therefore, these benchmarks are classified into two categories: *Small* and *Larger* benchmarks (shown in Table II). The *Small* benchmarks have simple enough control flow where the WC input data can be manually detected. Therefore, the WCET from the timing analyzer is close to the execution time obtained from the simulator. However, the WC input data is more difficult to manually detect for the *Larger* benchmarks. Therefore, the WCET from the timing analyzer may be much larger than the execution time obtained from the simulator for these larger benchmarks, since the observed cycles may not represent the execution of the WC paths. Table III 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 *Small* benchmarks.

In Table III, the *base* compilation time is the wall-clock time without performing code-positioning, while the *position* compilation time is the wall-clock time with WCET code positioning and 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

Table IV. Results after WCET Code Positioning and Target Alignment

	Before Positioning			After Positioning		After Alignment	
Program	Observed Cycles	WCET Cycles	WCET Ratio	WCET Cycles	Positioning Ratio	WCET Cycles	Alignment Ratio
bubblesort	7,372,782	7,623,795	1.034	7,622,295	1.000	7,497,546	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	794,487	819,785	1.010	814,424	0.906	801,700	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	557,223	582,126	1.084	578,024	0.932	569,419	0.924

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

Table IV shows the accuracy of our timing analyzer and 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 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 our timing analyzer. The *WCET cycles* should be larger than or equal to the *observed cycles*, since the WCET is the upper bound for the execution time and it should never be underestimated. The *ratios* show that these predictions are reasonably close for *Small* programs since it was not too difficult to determine the WC input data for *Small* programs or programs with few paths. For some *Larger* programs, it is harder to manually determine the WC input data, so the WCET from the timing analyzer is much larger than the *observed cycles* obtained from the simulator. This does not necessarily imply that the timing analyzer is inaccurate, but rather that the input data may be not executing the WC paths. We did not obtain the *observed cycles* after WCET positioning or WCET alignment since this would require new WCET input data due to changes in the WCET paths.

The results *after positioning* indicate the measurements taken after the WCET code-positioning algorithm described in Section 6 is applied immediately

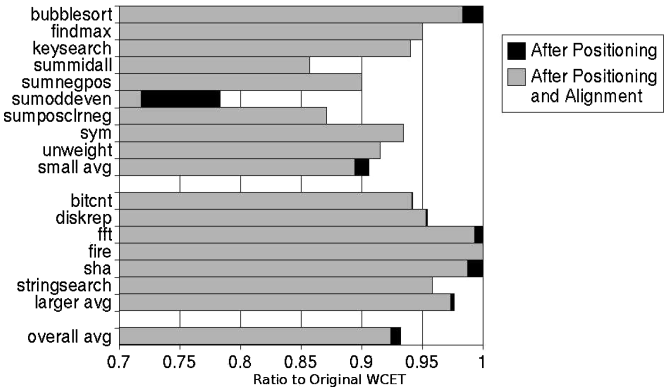

Fig. 16. Effect of WCET code positioning and alignment on WCET.

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 measurements that were obtained after the WCET target alignment algorithm in Section 7 is applied following WCET code positioning. There was often no benefit from target alignment since the misaligned targets, as depicted in Figure 3, do not occur that frequently. 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 because of the WCET target alignment, while three out of six *Larger* benchmarks improved, which resulted in over an additional 0.8% average reduction in WCET. Figure 16 shows in a chart the effect that WCET code positioning and alignment have on the WCET.

While the results in Table IV show a significant improvement in the predicted WCET, it would be informative to know if better positionings than those obtained by our greedy WCET code-positioning algorithm are possible. Like most benchmarks used for WCET prediction studies, the size of each *Small* benchmark is fairly small so that the WCET input data can be manually determined and the WCET observed cycles can be measured. The functions in these *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 V 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. The number of *permutations* varies, depending upon the number of *routines* in the benchmark and the number of basic blocks in each function.

Table V. Possible WCET Code Positioning Results for the *Small* Benchmarks

Program	Permutations	Minimum WCET	Greedy		Default		Maximum	
			WCET	Ratio	WCET	Ratio	WCET	Ratio
bubblesort	40,328	7,622,295	7,622,295	1.000	7,623,795	1.000	8,990,017	1.179
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	819,574	1.117	976,272	1.524

Unlike the measurements shown in Table IV, these WCET results exclude target-misprediction penalties. Our WCET positioning algorithm does not take target-misprediction penalties into account when making positioning decisions, since the WCET target-alignment optimization occurs after positioning. Thus, the WCETs are in general slightly lower than the WCETs, shown in Table IV.

The *minimum WCET* represents the lowest WCET found by performing the exhaustive search. There are typically multiple code positionings that result in an equal *minimum WCET*. We found that the *greedy WCET* obtained by our algorithm was always identical to the *minimum WCET* for each function in each benchmark for the *Small* test suite. It appears that our greedy algorithm is very effective at finding an efficient WCET code positioning and we anticipate that it would also work well on larger benchmarks.

The *default WCET* and *maximum WCET* are also given in Table V. The *default WCET* represents the WCET of the default code layout without code positioning. On average, the *default WCET* is 11.6% worse than the *minimum WCET*. The *maximum WCET* represents the highest WCET found during the exhaustive search. The results show that the *maximum WCET* is 50.2% higher, on average, than the *minimum WCET*. While the *default WCET* is relatively efficient compared to the *maximum WCET*, the *greedy WCET* still is a significant improvement over just using the default code positioning.

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

The effect on ACET after WCET code positioning and target alignment is shown in Table VI. The ACET cycles are obtained from the simulator when

Table VI. ACET Results After WCET Code Positioning and Target Alignment

	Baseline	After Positioning		After Alignment	
Program	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

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 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 one cycle. After code positioning, the execution time of the original WC path is reduced by three 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 one 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 three 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.

9. FUTURE WORK

We have performed code positioning and target-alignment optimizations based on the WC path information from the timing analysis. There is much future research that can be accomplished to enhance this code-positioning algorithm.

1. The WCET code-positioning algorithm could be used to improve the WCET for applications on other processors. Besides improving WC performance by reducing the transfers of control along the WC paths, this algorithm may be adapted to improve instruction cache performance along the WC path for processors with caches. Data cache performance may also be improved by this effort, although it may be more subtle.
2. We can attempt to reduce the compilation time. 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. The compiler and the timing analyzer are separate processes and exchange data via files. If we could merge the compiler and the timing analyzer into one process, it would speed up the algorithm.
3. Currently, the algorithm is automatic for benchmarks with bounded loops, since the timing analyzer can give exact clock cycles for the WCET of this kinds of programs. If the timing analyzer can produce WCET with a symbolic number of loop iterations as the parameters, then this algorithm can be modified to reduce the WCET for programs whose number of iterations cannot be statically determined by the compiler [Vivancos et al. 2001]. While the algorithm may not be able to determine which loop nests require the most cycles, the algorithm could be used to minimize the WCET within a loop nest.

10. CONCLUSIONS

In this paper, we have described a code-positioning algorithm that is driven by WCET path information from timing analysis, as opposed to ACET frequency data from profiling. WCET code positioning is more challenging than ACET code positioning, since the WC paths may change after changing the order of basic blocks. A WCET code-positioning optimization should attempt to minimize the maximum execution time among all the paths. Our algorithm addresses this issue by initially assuming that all edges are unpositioned. At each step, the algorithm conservatively estimates the WC paths in the function, based on the currently unpositioned edges and uses this information to select the next edge to make contiguous. The edges along the WC paths have highest priority to become contiguous to reduce the branch penalties. The paths that cannot contribute to the WCET are determined by calculating when their UB-WCET is less than the LB-WCET of another path in the same loop or outermost level of a function. Edges that appear in only noncontributing paths have the lowest priority for being made contiguous. We have implemented the algorithm and have demonstrated that it can improve the WCET of applications on a machine

with transfer of control penalties. In fact, our greedy WCET code-positioning algorithm obtains optimal results on the SC100 for tested programs with a small number of basic blocks. A related compiler optimization called WCET target alignment has also been implemented and evaluated. The target alignment optimization reduces WCET due to target-misalignment penalties. Thus, we have shown that it is feasible to develop specific compiler optimizations that are designed to improve WCET using WCET path information as opposed to improving ACET using frequency data. Code positioning determines the order of the basic blocks, but, in general, 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.

ACKNOWLEDGMENTS

We thank StarCore for providing the necessary software and documentation that were used in this project. This research was supported in part by NSF grants EIA-0072043, CCR-0208581, CCR-0208892, CCR-0310860, CCR-0312493, CCR-0312531, and CCR-0312695.

REFERENCES

ARNOLD, R., MUELLER, F., AND WHALLEY, D. 1994. Bounding worst-case instruction cache performance. In *Proceedings of the Fifteenth IEEE Real-time Systems Symposium*, San Juan. IEEE Computer Society Press. 172–181.

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

BENITEZ, M. E. AND DAVIDSON, J. W. 1988. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language design and Implementation*, Atlanta, GA. ACM Press, New York. 329–338.

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

CALDER, B. AND GRUNWALD, D. 1994. Reducing branch costs via branch alignment. In *Proceeding of ASPLOS'94*, San Jose, CA. ACM Press, New York. 242–251.

ENGBLOM, J. AND ERMEDAHL, A. 2000. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-time System Symposium*, Orlando, FL. IEEE Computer Society Press, 875–889.

EYRE, J. AND BIER, J. 1998. Dsp processors hit the mainsteam. *IEEE Computer 31*, 8 (Aug.), 51–59.

HARMON, M., BAKER, T., AND WHALLEY, D. 1994. A retargetable technique for prediction execution time of code segments. *Real-Time Systems.* 159–182.

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

HEALY, C. AND WHALLEY, D. 2000. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transaction on Software Engineering 28*, 8 (Aug.), 763–781.

HEALY, C., WHALLEY, D., AND HARMON, M. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, Pisa. IEEE Computer Society Press. 288–297.

HEALY, C., ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. 1999. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers 48*, 1 (Jan.), 53–70.

HEALY, C., SJODIN, M., RUSTAGI, V., WHALLEY, D., AND VAN ENGELEN, R. 2000a. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems 18*, 2 (May), 121–148.

HEALY, C., WHALLEY, D., AND VAN ENGELEN, R. 2000b. A general approach for tight timing predictions of non-rectangular loops. In *WIP Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Washington, DC. IEEE Computer Society Press. 11–14.

HONG, S. AND GERBER, R. 1993. Compiling real-time programs into schedulable code. In *Proceedings of the SIGPLAN'93*, Albuquerque, NM. ACM Press, New York. 166–176.

KO, L., HEALY, C., RATLIFF, E., ARNOLD, R., WHALLEY, D., AND HARMON, M. 1996. Supporting the specification and analysis of timing constraints. In *Proceeding of the IEEE Real-Time Technology and Application Symposium*, Boston, MA. IEEE Computer Society Press. 170–178.

KO, L., AL-YAQOUBI, N., HEALY, C., RATLIFF, E., ARNOLD, R., WHALLEY, D., AND HARMON, M. 1999. Timing constraint specification and analysis. *Software Practice & Experience 29*, 1 (Jan.), 77–98.

KULKARNI, P., ZHAO, W., MOON, H., CHO, K., WHALLEY, D., DAVIDSON, J., BAILEY, M., PAEK, Y., AND GALLIVAN, K. 2003. Finding effective optimization phase sequences. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA. ACM Press, New York. 12–23.

LEE, S., LEE, J., PARK, C., AND MIN, S. 2004. A flexible tradeoff between code size and wcet using a dual instruction set processor. In *International Workshop on Software and Compilers for Embedded Systems*, Amsterdam. Springer, New york. 244–258.

LI, Y., MALIK, S., AND WOLFE, A. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the Sixteenth IEEE Real-time Systems Symposium*, Pisa. IEEE Computer Society Press. 298–307.

LIM, S., BAE, Y., JANG, G., RHEE, B., MIN, S., PARK, C., SHIN, H., PARK, K., AND KIM, C. 1994. An accurate worst case timing analysis technique for risc processors. In *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, San Juan. IEEE Computer Society Press. 875–889.

LUNDQVIST, T. AND STENSTROM, P. 1998. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems(LCTES'98)*, Montreal. IEEE Computer Society Press. 1–15.

MCFARLING, S. AND HENNESSY, J. 1986. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, Tokyo, Japan. 396–403.

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

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

PETTIS, K. AND HANSEN, R. 1990. Profile guided code position. In *Proceeding of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, ACM Press, New York. 16–27.

SHAW, A. C. 1989. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering 15*, 7, 875–889.

STAR CORE, I. 2001a. Sc100 simulator reference manual.

STAR CORE, I. 2001b. Sc110 dsp core reference manual.

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

VIVANCOS, E., HEALY, C., MUELLER, F., AND WHALLEY, D. 2001. Parametric timing analysis. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, Snowbird, UT. ACM Press, New York. 83–93.

WHITE, R. T., MUELLER, F., HEALY, C., WHALLEY, D., AND HARMON, M. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, Montreal. IEEE Computer Society Press. 192–202.

WHITE, R., MUELLER, F., HEALY, C., WHALLEY, D., AND HARMON, M. 1999. Timing analysis for data caches and wrap-around-fill caches. *Real-Time Systems 17*, 1 (Nov.), 209–233.

ZHAO, W., CAI, B., WHALLEY, D., BAILEY, M., VAN ENGELEN, R., YUAN, X., HISER, J., DAVIDSON, J., GALLIVAN, K., AND JONES, D. 2002. Vista: A system for interactive code improvement. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, Berlin. ACM Press, New York. 155–164.

Received May 2005; revised November 2005; accepted November 2005