Tuning the WCET of Embedded Applications

Wankang Zhao[1], Prasad Kulkarni[1], David Whalley[1],
Christopher Healy[2], Frank Mueller[3], Gang-Ryung Uh[4]

[1]Computer Science Dept., Florida State University, Tallahassee, FL 32306-4530; e-mail: whalley@cs.fsu.edu

[2]Computer Science Dept., Furman University, Greenville, SC 29613; e-mail: chris.healy@furman.edu

[3]Computer Science Dept., North Carolina State University, Raleigh, NC 27695; e-mail: mueller@cs.ncsu.edu

[4]Computer Science Dept., Boise State University, Boise, ID 83725; e-mail: uh@cs.boisestate.edu

Abstract

It is advantageous to not only calculate the WCET of an application, but to also perform transformations to reduce the WCET since an application with a lower WCET will be less likely to violate its timing constraints. In this paper we describe an environment consisting of an interactive compilation system and a timing analyzer, where a user can interactively tune the WCET of an application. After each optimization phase is applied, the timing analyzer is automatically invoked to calculate the WCET of the function being tuned. Thus, a user can easily gauge the progress of reducing the WCET. In addition, the user can apply a genetic algorithm to search for an effective optimization sequence that best reduces the WCET. Using the genetic algorithm, we show that the WCET for a number of applications can be reduced by 7% on average as compared to the default batch optimization sequence.

1. Introduction

Generating acceptable code for applications 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 constraints are not met, even only occasionally in a hard real-time system, then the system may not be considered functional. The *worst-case execution time* (WCET) must be calculated to determine if a timing constraint will be met.

Unfortunately, many embedded system developers empirically estimate the WCET by testing the application and measuring the execution time. Testing alone is unsafe since the WC input data is often difficult to derive. This approach can result in an unsafe application since WC timing constraints may not be met when an application is deployed. More knowledgeable developers will test, measure, and make conservative assumptions in case the timing measurements do not truly reflect the WCET, which can result in loose estimates and higher overall costs. Thus, accurate WCET predictions are required to produce safe and cost effective embedded systems. Accurate WCET predictions can only be obtained by a tool that statically analyzes an application to calculate the WCET. Such a tool is called a *timing analyzer*, and the process of performing this calculation is called *timing analysis*.

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. In contrast, conservative assumptions concerning WCET may result in a processor being deployed that has a higher clock rate and consumes more power.

Automatically generating acceptable code for embedded microprocessors with a compiler is often much more difficult than generating code for general-purpose processors. Besides sometimes having to meet a variety of conflicting constraints, embedded microprocessors are typically much less regular and have many specialized architectural features. Because of the typical large volumes produced for a product involving an embedded computer system, many embedded systems applications are still being developed in assembly language by hand in order to meet the imposed constraints and to deal with the difficulty of exploiting the features of the machine. In fact, two of the authors of this paper have recently spent time in industry and have personally witnessed the development and maintenance of assembly code applications. However, developing an application in assembly has many disadvantages that include higher development and maintenance costs and less portable code.

It would be desirable to develop embedded system applications in a high level language and still be able to tune the WCET of an application. We have provided this capability by integrating a WCET timing analyzer with an interactive compilation system called VISTA (Vpo Interactive System for Tuning Applications) [1, 2]. One feature of VISTA is that it can automatically obtain

performance feedback information, which can be used by both the application developer and the compiler to make phase ordering decisions. This information can include a variety of measures, such as execution time or code size. In this paper we describe how we modified VISTA so it can use WCET as one of its performance criteria.

The remainder of the paper is structured as follows. First, we review related work on improving, displaying, and estimating WCET. Second, we give a brief overview of the timing analyzer that we used in this work. Third, we summarize the StarCore SC100 and how we retargeted our compiler and timing analyzer for this processor. Fourth, we describe VISTA and how we integrated the timing analyzer with this framework. Fifth, we show the benefits that were achieved by performing searches using a genetic algorithm to improve the WCET. Finally, we discuss future plans for developing compiler optimizations to improve WCET and give the conclusions of the paper.

2. Related Work

There have been a variety of different techniques used for timing analysis of optimized code over the years [3, 4, 5, 6, 7]. However, we are unaware of any timing analyzer whose predictions are used by a compiler to select which optimizations should be applied.

While there has been much work on developing compiler optimizations to reduce execution time and, to a lesser extent, compiler optimizations to reduce space and power consumption, there has been very little work where compiler optimizations have been developed to reduce WC performance. Marlowe and Masticola outlined how a variety of standard compiler optimizations could potentially affect timing constraints of critical portions in a task. However, no implementation was described [8]. Hong and Gerber developed a programming language with timing constructs and used a trace scheduling approach to improve code in what would be deemed a critical section of the program. 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 [9]. 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 entire tasks to determine if a schedule can be met. Lee *et. al.* used WCET information to choose how to generate code on a dual instruction set processor for the ARM and the Thumb [10]. ARM code is generated for a selected subset of basic blocks that can impact the WCET. Thumb code is generated for the remaining blocks to minimize code size. In contrast, we are using WCET information to select compiler optimizations, as opposed to which instruction set to select for code generation.

A user interface was developed at Florida State University that allows users to select portions of source code and obtain timing predictions. Unlike VISTA, this interface did not allow the user to affect the generated code or provide feedback during the compilation process [11, 12, 13].

Genetic algorithms have long been used to search for solutions in a space that is too large to exhaustively evaluate. Genetic algorithms have been used to search for effective optimization sequences to improve speed, space, or a combination of both [14, 2]. 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) [15, 16, 17]. Our approach, in contrast, relies on a genetic algorithm to identify optimization phase sequences that result in reduced WCET, which is an orthogonal problem.

3. The Timing Analyzer

In this section we briefly describe the timing analyzer that we have previously developed and that served as the starting point for the timing analyzer in this study. Figure 1 depicts the organization of the framework that was used by the authors in the past to make WCET predictions. The VPO (Very Portable Optimizer) compiler [18] was modified to produce the control flow and constraint information as a side effect of the compilation of a source file. 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 uses the control-flow and constraint information, caching categorizations, and machine-dependent information (e.g. pipeline characteristics) to make the timing predictions.

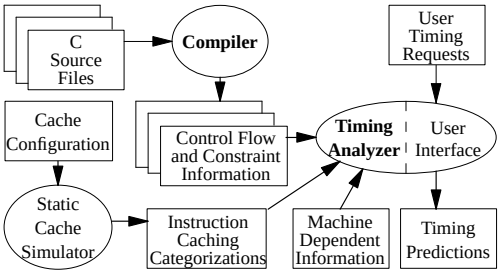

Figure 1: Overview of the Existing Process to Obtain WCET Predictions

The timing analyzer calculates the WCET for each function and loop in the program. It performs this analysis in a bottom up fashion, where the WCET for an inner

loop (or called function) is calculated before determining the WCET for an outer loop (or calling function). The WCET information for an inner loop (or called function) is used when it is encountered in an outer-level path.

Besides addressing architectural features, such as caching [19, 3, 20, 21, 22, 23] and pipelining [24, 3], the timing analyzer also automatically detects control-flow constraints. One type of constraint is the maximum iterations associated with each loop, including nonrectangular loop nests [25, 26, 27]. Another constraint type is when a branch will be taken or fall through. The timing analyzer uses these constraints to detect infeasible paths through the code or how often a given path can be executed [28, 4].

4. Porting to the SC100

In order to determine the effectiveness of improving the WCET for applications on an embedded processor, we ported both the VPO compiler and the timing analyzer to the StarCore SC100 processor [29]. In the past we had made WCET predictions for the MicroSPARC I, which is a general-purpose processor [30]. We were able to produce very tight WCET predictions with respect to a MicroSPARC I simulator that we had developed. Unfortunately, it is very difficult to produce cycle-accurate simulations for a general-purpose processor due to complexity of its memory hierarchy and its interaction with an operating system that can cause execution times to vary. Unlike the MicroSPARC I, the SC100 has neither a memory hierarchy (no caches or virtual memory system) nor an OS [29]. In addition, we were able to obtain a simulator for the SC100 from StarCore [31]. 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.

Some of the general features of the SC100 are as follows. The SC100 has no architectural support for floating-point operations since it is a digital signal processor and was designed instead for fixed-point arithmetic. It has 16 data registers and 16 address registers. The 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 has a simple five stage pipeline, where most instructions can execute in a single stage. There are no pipeline interlocks. It is the compiler's responsibility to insert noop instructions to delay a subsequent instruction that uses the result of a preceding instruction when the result will not be available in the pipeline. Transfers of control (taken branches, unconditional jumps, calls, returns) result in a one to three cycle penalty depending on the addressing mode used and if a transfer of control uses a delay slot.

There were several modifications we made to support timing analysis of applications compiled for the SC100. First, we modified the machine-dependent information (see Figure 1) to indicate how instructions proceed through the SC100 pipeline. We had to identify the instructions that require extra cycles in the pipeline. For instance, if a memory addressing mode on the SC100 performs an arithmetic calculation, then one additional one cycle is required. Second, we also updated the timing analyzer to treat all cache accesses as hits since instructions and data on the SC100 can in general be accessed in a single cycle from ROM and RAM, respectively. Thus, the static cache simulation step shown in Figure 1 is now bypassed for the SC100. Third, we had to modify the timing analyzer to address the penalty for transfers of control. When calculating the WCET of a path, we had to determine if each conditional branch in the path was taken or fell through since untaken branches are not assessed this penalty. In addition, we had to determine the size of each instruction and its alignment in memory. SC100 instructions are grouped into fetch sets, which are four words (eight bytes) in size. Transferring control to an instruction in a new fetch set that spans more than one fetch set results in an additional cycle delay.

We have found that transfer of control penalties can lead to nonintuitive WCET results. For instance, consider the flow graph in Figure 2. A superficial inspection would lead one to believe that the path 1→2→3 is the WCET path through the graph. However, if the taken branch penalty in the path 1→3 outweighs the cost of executing the instructions in block 2, then 1→3 would be the WCET path. This simple example illustrates the importance of using a timing analyzer to calculate the WCET. 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.

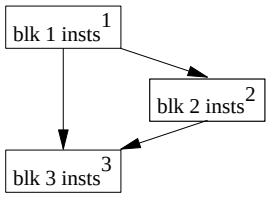

Figure 2: Example Control-Flow Graph

Measurements indicating the accuracy of the WCET predictions produced by our timing analyzer will be shown later in the paper. In general, we could produce fairly accurate WCET predictions since some of the more problematic issues, which include memory hierarchies and operating systems, are not present on this processor.

5. Integrating with VISTA

This section provides a brief overview of the VISTA framework used for tuning the WCET of applications. We also describe the modifications that were required to integrate our timing analyzer with VISTA so that the current WCET can be presented to the user and can be used by the compiler when tuning an application.

The flow of information is depicted in Figure 3, which includes the VPO compiler, a viewer, and the timing analyzer described in Section 3. The programmer initially indicates a source file to be compiled and then specifies requests through the viewer, which include the order and scope of the optimization phases to be applied. After applying each optimization phase, the compiler sends information about the current instructions, control flow, and constraint information to the timing analyzer and the timing analyzer sends its WCET predictions back to the compiler. The user is presented with the state of the requested performance criteria, which for this version of VISTA includes the WCET and the code size.

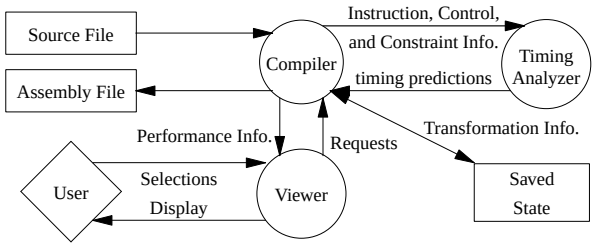

Figure 3: Overview of Tuning WCET in VISTA

In previous versions of VISTA, the compiler obtained dynamic measurements after applying each optimization phase by instrumenting the code, producing the assembly code, linking and executing the program, and getting performance measures from the execution [2]. Since we used representative input data to obtain this dynamic measure, we were in effect obtaining *average case execution time* (ACET) information.

Figure 4 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 the improvements in WCET and code size are shown. Thus, a user can easily gauge the progress that has been made at tuning the current function.

Besides applying predefined compiler optimization phases, an application developer can also specify transformations manually by inserting, modifying, and deleting instructions. Upon request the system also answers queries, such as which registers are live at a specific point in the program representation. This information can assist the developer to make safe and effective manual transformations. The ability to specify transformations manually is useful for exploiting special-purpose hardware that currently cannot be automatically exploited by the compiler.

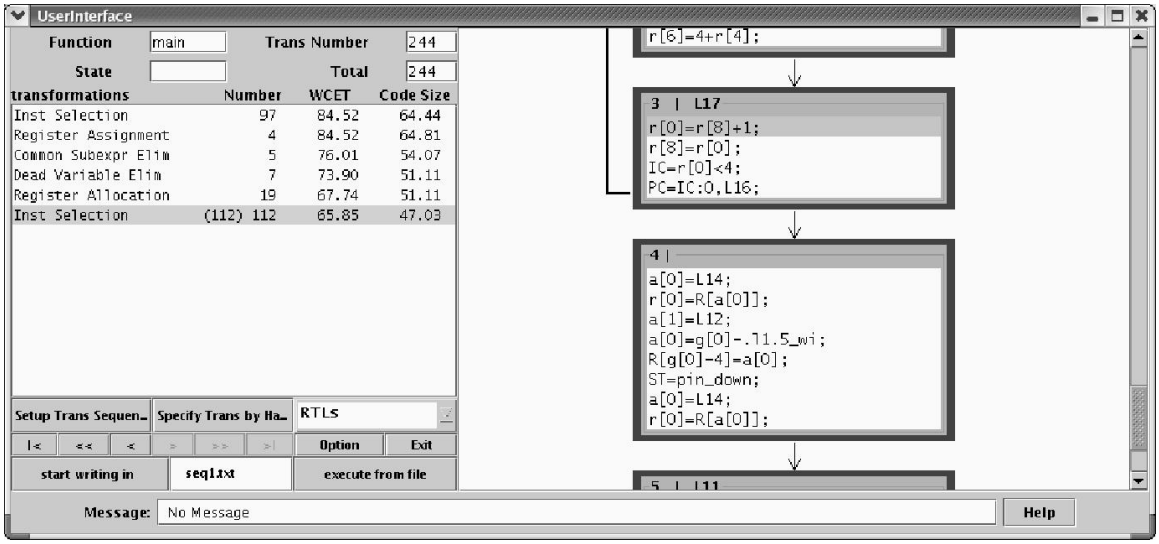

Figure 4: Main Window of VISTA Showing History of Optimization Phases

The user also has the ability to reverse previously applied transformations, which supports experimentation when tuning an application. This is accomplished by writing the sequence of applied transformations to a file. Afterwards, VISTA reads in the intermediate code generated by the front end and applies the list of transformations to generate the program representation that was previously produced in the compilation. The sequence of applied transformations is also written to a file when the user chooses to complete the tuning of a function or terminate the session. This file is automatically read when VISTA is later invoked so these transformations can be reapplied at a later time, enabling future updates.

There are some initial actions that are performed by VISTA before a function can have its WCET tuned. The information to be sent to the timing analyzer from the compiler includes the number of loop iterations. The compiler detects this information as a side effect of performing a number of optimizations. Thus, VISTA was modified to automatically perform a set of optimizations when a function is being compiled for the first time that allows the compiler to calculate this information. The code-improving transformations are then automatically reversed. In a second pass, the compiler performs the compulsory phases, which includes *register assignment* (assigning pseudo registers to hardware registers) and *fix entry/exit* (inserting instructions to manage the run-time stack). The compiler emits the information and the timing analyzer is invoked to obtain the baseline WCET for each function. At this point VISTA can be used to tune the WCET for each function within the application.

VISTA also allows a user to specify a set of distinct optimization phases and have the compiler attempt to find the best sequence for applying these phases. Figure 5 shows the different options that VISTA provides the user to control the search. The user specifies the *sequence length*, which is the total number of phases applied in each sequence. We performed a set of experiments described in the next section that use the *biased sampling search*, which applies a genetic algorithm in an attempt to find the most effective sequence within a limited amount of time since in many cases the search space is too large to exhaustively evaluate [32]. The genetic algorithm treats each optimization phase as a gene and each sequence of phases as a 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 limits 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.

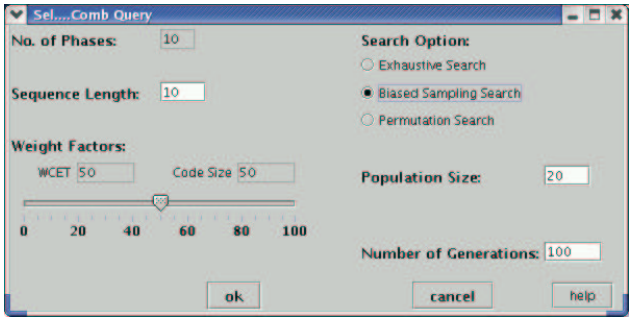

Figure 5: Selecting Options to Search for Sequences

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 6 shows a snapshot of the status of the search that was selected in Figure 5. 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.

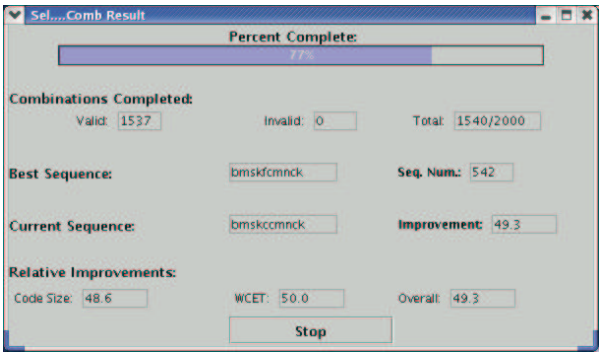

Figure 6: Window Showing the Search Status

6. 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. Table 1 shows the benchmarks and applications we used for our experiments. These include a subset of the *DSPstone* fixed-point kernel benchmarks[1] and other DSP benchmarks or programs that we have used

[1] The only DSPstone fixed-point kernel benchmarks we did not include were those that could not be automatically processed by our timing analyzer. In particular, the number of iterations for loops in some benchmarks could not be statically determined by our compiler. While our framework allows a user to interactively supply this information, we excluded such programs to facilitate automating the experiments.

Category	Program	Description
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	fft	128 point complex FFT
	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
	sym	tests if a 100x100 matrix is symmetric

Table 1: Benchmarks Used in the Experiments

in previous studies. Many DSP benchmarks represent kernels of applications where most of the cycles occur. Such kernels in DSP applications have been historically optimized in assembly code by hand to ensure high performance [33]. In contrast, all of the results in this section are from code that was automatically generated by VISTA.

Note that the *DSPstone* fixed-point kernel benchmarks are small and do not have conditional constructs, such as `if` statements. The *other* benchmarks shown in Table 1 were selected since they do have conditional constructs, which means the WCET and ACET input data may not be the same.

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. The analysis time required for our timing analyzer is proportional to the number of unique paths at each loop and function level in the program. In contrast, tuning for ACET typically takes much longer since the simulation time of the SC100 simulator is proportional to the number of instructions executed. We found that the average time required to tune the WCET of each function in our experiments was about 25 minutes and this would have taken several hours if we had used simulation.

Table 2 shows each of the candidate code-improving phases that we used in the experiments when tuning each function with the genetic algorithm. 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, we perform 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. Finally, we also perform additional code-improving phases after the sequence, such as *instruction scheduling*. For the SC100 another compulsory phase is required to insert noops when pipeline constraints need to be addressed.

Our genetic algorithm searches were accomplished in the following manner. We set the sequence (chromosome) length to be 1.25 times the number of phases that successfully applied one or more transformations by the batch compiler for the function. We felt this was a reasonable limit and gives us 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. We set the population size (fixed number of sequences or chromosomes) to twenty and each of these initial sequences is randomly initialized with candidate optimization phases. We performed 200 generations when searching for the best sequence for each function. We sort the sequences in the population by a *fitness value* based on the WCET produced by the timing analyzer and/or code size. At each generation (time step) we remove the worst sequence and three others from the lower (poorer performing) half of the population chosen at random. Each of the removed sequences are replaced by randomly selecting a pair of the remaining sequences from the upper half of the population

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.

Table 2: Candidate Optimization Phases in the Genetic Algorithm Experiments

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 we find identical sequences in the same population, then we replace the redundant sequences with ones that are randomly generated. The characteristics of this genetic algorithm search are very similar to those used in past studies, [14, 2] except the objective function now is minimizing the WCET.

Table 3 shows the WCET prediction results for the benchmarks in Table 1. The *batch sequence* results are those that are obtained from the sequence of applied phases when we use 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 [2]. The *observed cycles* were obtained from running the compiled programs through the SC100 simulator. All input and output were accomplished by reading from and writing to global variables to avoid having to estimate the WCET of performing actual I/O. The *WCET cycles* are the WCET predictions obtained from our timing analyzer. The *ratio*s show that these predictions are reasonably close to the actual WCET.[2] The *ratios* for the *best sequence from GA* results in Table 3 are similar, but the code being measured was the best sequence found by the genetic algorithm. The *WCET GA to WCET 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. We found that the average number of generations to find the best sequence was 51 out of the 200 generations attempted. 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. While there were some aberrations due the randomness of using a genetic algorithm, most of the benchmarks had improved WCETs. The WCET cycles decreased by 6.6% on average. This illustrates the benefit of using a genetic algorithm to search for effective optimization sequences to improve WCET.

[2] There are still small some overestimations that we need to address. This problem is exacerbated due to not having access to the source code of the SC100 simulator and the simulated pipeline behavior not always exactly matching the behavior described in the SC100 documentation.

Category	Program	Batch Sequence			Best Sequence from GA			WCET GA to WCET Batch Ratio
		Observed Cycles	WCET Cycles	Ratio	Observed Cycles	WCET Cycles	Ratio	
DSPstone	convolution	683	691	1.012	619	627	1.013	**0.907**
	complex_update	152	158	1.039	149	155	1.040	**0.981**
	dot_product	121	132	1.091	110	119	1.082	**0.902**
	fir	1133	1140	1.006	1004	1012	1.008	**0.888**
	fir2dim	5809	6100	1.050	5430	5668	1.044	**0.929**
	iir_biquad_one_section	133	140	1.053	130	137	1.054	**0.979**
	iir_biquad_N_sections	1175	1194	1.016	1282	1297	1.012	**1.086**
	lms	1599	1609	1.006	1259	1269	1.008	**0.789**
	matrix	39213	39668	1.012	35624	35976	1.010	**0.907**
	matrix_1x3	274	291	1.062	260	274	1.054	**0.942**
	n_complex_updates	2869	2875	1.002	2821	2826	1.002	**0.983**
	n_real_updates	1698	1705	1.004	1442	1449	1.005	**0.850**
	real_update	81	88	1.086	85	90	1.059	**1.023**
other	fft	78128	78645	1.007	61572	61634	1.001	**0.784**
	summidall	19508	19515	1.000	18510	18516	1.000	**0.949**
	summinmax	24011	24017	1.000	22010	22016	1.000	**0.917**
	sumnegpos	20010	20015	1.000	19011	19018	1.000	**0.950**
	sumoddeven	22021	23045	1.046	22023	22045	1.001	**0.957**
	sym	223366	228125	1.021	218416	223076	1.021	**0.978**
	average	23262	23639	1.027	21724	22011	1.022	**0.934**

Table 3: WCET Prediction Results

Category	Program	optimizing for WCET		optimizing for space		optimizing for both		
		effect on WCET	effect on size	effect on WCET	effect on size	effect on WCET	effect on size	avg effect on both
DSPstone	convolution	**0.907**	0.956	0.907	**0.956**	0.907	0.956	**0.931**
	complex_update	**0.981**	0.964	1.272	**0.982**	0.981	0.982	**0.982**
	dot_product	**0.902**	0.927	0.902	**0.927**	0.902	0.927	**0.914**
	fir	**0.888**	0.948	1.149	**0.916**	0.889	0.948	**0.918**
	fir2dim	**0.929**	0.976	0.930	**0.906**	0.930	0.906	**0.918**
	iir_biquad_one_section	**0.979**	0.981	1.021	**1.000**	0.979	0.981	**0.980**
	iir_biquad_N_sections	**1.086**	1.000	1.226	**0.991**	1.136	0.991	**1.063**
	lms	**0.789**	0.926	0.789	**0.921**	0.890	0.955	**0.922**
	matrix	**0.907**	0.929	0.997	**0.996**	0.967	0.969	**0.968**
	matrix_1x3	**0.942**	0.846	0.942	**0.846**	0.942	0.846	**0.894**
	n_complex_updates	**0.983**	1.000	1.033	**0.966**	0.994	0.990	**0.992**
	n_real_updates	**0.850**	0.949	0.850	**0.949**	0.850	0.949	**0.899**
	real_update	**1.023**	1.014	1.023	**1.014**	0.898	0.942	**0.920**
other	fft	**0.784**	0.942	0.788	**0.913**	0.775	0.910	**0.842**
	summidall	**0.949**	1.021	0.949	**1.021**	0.949	1.021	**0.985**
	summinmax	**0.917**	0.889	1.333	**0.857**	0.917	0.889	**0.903**
	sumnegpos	**0.950**	1.133	1.000	**1.022**	1.000	1.000	**1.000**
	sumoddeven	**0.957**	1.134	1.305	**1.000**	0.979	1.015	**0.997**
	sym	**0.978**	0.961	0.999	**0.931**	1.000	0.971	**0.985**
	average	**0.934**	0.967	1.022	**0.953**	0.941	0.954	**0.948**

Table 4: Effect on WCET and Code Size Using the Three Fitness Criteria

In addition to improving WCET, we thought it would be interesting to see the improvement in code size. Table 4 shows the results obtained for each benchmark by applying the genetic algorithm when changing the fitness criteria. For each benchmark we performed three different searches, which are based on WCET only (*optimizing for WCET*), code size only (*optimizing for space*), and 50% for each factor (*optimizing for both*). For each type of search, we show the effect both on WCET and on code size. The results that are supposed to improve according the specified fitness criteria used are shown in boldface. For these results, the genetic algorithm was able to typically find a sequence for each benchmark that either achieves the same result or obtains an improved result as

compared to the batch compilation. The results when optimizing for both WCET and code size showed that we were able to achieve a better overall benefit when both WCET and code size are considered.

7. Future Work

There is much future research that can be accomplished on tuning the WCET of embedded applications. We can vary the characteristics of the genetic algorithm search. It would be interesting to see the effect on a search as one changes aspects of the genetic algorithm, such as the sequence length, population size, number of generations, etc. In addition, it would be interesting to perform searches involving more aggressive compiler optimizations, different benchmarks, and different processors.

Now that we have integrated a timing analyzer with a compiler, there are a number of compiler optimizations that we plan to develop with the goal of reducing the WCET. These optimizations will use the WCET path information provided by the timing analyzer to drive the optimizations that will be performed.

8. Conclusions

There are several contributions that we have presented in this paper. First, we have 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 easily gauge the progress that has been made. To the best of our knowledge, we believe this is the first compiler that interacts with a timing analyzer to use WCET predictions during the compilation of applications. Second, we have shown that the 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, we have shown that both WCET and code size improvements can be simultaneously obtained. Both of these criteria are important factors when tuning applications for an embedded processor.

9. Acknowledgements

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

10. References

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

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

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

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

[5] J. Engblom, P. Altenbernd, and A. Ermedahl, "Facilitating Worst-Case Execution Time Analysis for Optimized Code," *Euromicro Workshop on Real-Time Systems*, pp. 146-153 (June 1998).

[6] S. Lim, J. Kim, and S. Min, "A Worst Case Timing Analysis Technique for Optimized Programs," *International Conference on Real-Time Computing Systems and Applications*, pp. 151-157 (October 1998).

[7] R. Kirner and P. Puschner, "Transformation of Path Information for WCET Analysis during Compilation," *Euromicro Conference on Real-Time Systems*, (June 2001).

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

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

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

[11] L. Ko, D. B. Whalley, and M. G. Harmon, “Supporting User-Friendly Analysis of Timing Constraints,” *Proceedings of the ACM SIGPLAN Notices 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems* **30**(11) pp. 99-107 (November 1995).

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

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

[14] K. Cooper, P. Schielke, and D. Subramanian, “Optimizing for Reduced Code Space Using Genetic Algorithms,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1-9 (May 1999).

[15] 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 1998).

[16] 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 Technology and Applications Symposium*, pp. 179-188 (June 1998).

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

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

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

[20] F. Mueller, “Timing Predictions for Multi-Level Caches,” *Proceedings of the ACM SIGPLAN Notices 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pp. 29-36 (June 1997).

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

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

[23] F. Mueller, “Timing Analysis for Instruction Caches,” *Real-Time Systems* **18**(2) pp. 209-239 (May 2000).

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

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

[26] C. A. Healy, R. van Engelen, and D. B. Whalley, “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 (June 1999).

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

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

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

[30] Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.

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

[32] J. Holland, *Adaptation in Natural and Artificial Systems*1989.

[33] J. Eyre and J. Bier, “DSP Processors Hit the Mainstream,” *IEEE Computer* **31**(8) pp. 51-59 (August 1998).