WCET-aware Register Allocation based on Graph Coloring[*]

Heiko Falk
Computer Science 12
Technische Universität Dortmund
D - 44221 Dortmund, Germany
Heiko.Falk@tu-dortmund.de

ABSTRACT

Current compilers lack precise timing models guiding their built-in optimizations. Hence, compilers apply ad-hoc heuristics during optimization to improve code quality. One of the most important optimizations is register allocation. Many compilers heuristically decide when and where to spill a register to memory, without having a clear understanding of the impact of such spill code on a program's run time.

This paper extends a graph coloring register allocator such that it uses precise worst-case execution time *(WCET)* models. Using this WCET timing data, the compiler tries to avoid spill code generation along the critical path defining a program's WCET. To the best of our knowledge, this paper is the first one to present a WCET-aware register allocator. Our results underline the effectiveness of the proposed techniques. For a total of 46 realistic benchmarks, we reduced WCETs by 31.2% on average. Additionally, the runtimes of our WCET-aware register allocator still remain acceptable.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers; Optimization; C.3 [**Real-time and embedded systems**]; B.3.3 [**Memory Structures**]: Worst-case analysis

General Terms

Algorithms, Performance

Keywords

WCET, Register Allocation

1. INTRODUCTION

Embedded systems are often real-time systems whose correctness depends on both the logical results and on the time at which the results are produced. A program's *worst-case execution time (WCET)* is used to guarantee that real-time constraints are safely met. But besides safety, the market demands high performance, energy efficiency and low cost. Hence, designing such products implies solving a complex optimization problem with multiple optimization criteria. Compilers play an important role during real-time system design since they are able to apply automated optimizations improving the quality of the generated executable code.

Unfortunately, even modern optimizing compilers are often unable to quantify the effect of an optimization since they lack precise timing models [8]. Hence, simple ad-hoc heuristics are applied during optimization in the hope that these finally improve code quality. But it is well-known that this is not always true: due to the absence of precise models, optimizations may have a negative impact on code quality.

Among all optimizations studied in the past, *register allocation* is considered the most important one. It intends to use a processor's registers most efficiently in order to minimize slow main memory accesses. Due to the increasing speed gap between processors and memories, accesses to physical processor registers *(PHREGs)* are orders of magnitudes faster than memory accesses. However, memory accesses can not be totally avoided during register allocation, since the amount of temporary variables (aka. *virtual registers, VREGs*) at a certain place in a program can exceed the number of available PHREGs. In such a situation, *spill code* is inserted swapping a register out to memory and back.

Currently, register allocators usually decide heuristically where to insert spill code. Due to a lack of precise models, the compiler is unaware of the impact of generated spill code on a program's execution time. Especially in the area of real-time system design and optimization, badly placed spill code can have a dramatic impact on a program's WCET.

This paper is the first one to present a technique for a WCET-aware graph coloring register allocator. The main contributions of the proposed register allocator are that it

- explicitly uses WCET data during optimization,
- automatically updates its WCET data in the course of the optimization in order to cope with the inherent instability of the critical path defining the WCET,
- reduces average WCETs by 31.2% for 46 benchmarks while requiring only moderate optimization runtimes.

Section 2 gives a survey of related work on register allocation, WCET optimization and compilers using formal models during optimization. Section 3 presents graph coloring which is the standard technique for register allocation, followed by the WCET-aware register allocator in Section 4. Section 5 describes the benchmarking results, and Section 6 summarizes this paper and gives an outlook on future work.

[*]Funded by the European Community's 7th Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

2. RELATED WORK

Graph coloring is the standard technique for register allocation nowadays. Due to its importance for compilers and for this paper, it is discussed in more detail in Section 3.

An optimal register allocator using *integer-linear programming (ILP)* is presented by [7]. As opposed to the motivation given in Section 1, [7] does not apply any heuristic during register allocation since ILP produces optimal results. The ILP minimizes code size, i. e. the total amount of generated spill code. Since code size is a criterion not requiring sophisticated models in a compiler, optimal results can be produced. Nevertheless, the absence of timing models implies that the impact of [7] on WCETs is fully unknown. How to make this ILP WCET-aware is unclear up to now.

A third standard register allocation technique is *linear scan* [11]. Using a linear order of all operations of a program, a life time interval is computed for each VREG. Register allocation is done by mapping each life time interval to a PHREG and by applying a simple spill heuristic if all PHREGs are in use. Compared to graph coloring, linear scan is fast but produces results of inferior quality. In addition, the code quality resulting from linear scan heavily depends on the chosen operation order and on the spill heuristic. Hence, linear scan allocation is a typical example of an optimization not guided by actual models.

Profile feedback optimization is discussed in [10] as a workaround for lacking formal models. Here, information about a program's run-time behavior is supplied to the compiler, which is collected by applying code instrumentation and run-time profiling. The profile-based register allocator of the Sun Studio compiler finally uses basic block counts from this profile data. In contrast to such profile-based approaches, our approach relies on timing models tightly integrated in a compiler making profiling runs obsolete.

One of the very few compilers featuring an actual model used during optimization is the energy-aware C compiler *encc* [13]. Its energy model reflects the energy dissipation of each operation of the ARM thumb instruction set, plus the energy consumed during various memory accesses. Using this energy model, *encc* exploits memory hierarchies by moving program code or data onto scratchpad memories and minimizes a program's energy consumption.

The compiler WCC [5] is the first fully functional compiler explicitly designed for WCET minimization. WCET timing models are integrated into WCC by coupling its backend with the static WCET analyzer aiT [1]. This way, WCC can apply static WCET analysis while optimizing and can use all the WCET-related data computed by aiT for optimization. WCC serves as technical infrastructure for the WCET-aware register allocator presented in this paper.

Compiler optimizations minimizing WCET are an emerging area of research where only few related works currently exist. In [9], a combination of procedure cloning and procedure positioning to improve worst-case I-cache performance is proposed. The authors of [4, 6, 12] propose to move parts of a program's code and data onto a scratchpad memory or onto a software-controlled cache. These papers focus on exploiting memory hierarchies outside a processor core to minimize WCETs. Exploiting the register file – which is that part of a memory hierarchy being closest to a processor – in a WCET-aware way has not yet been considered in any publication to the best of our knowledge.

3. TRADITIONAL GRAPH COLORING

Due to its relevance, traditional graph coloring is presented in-depth in this section. The overall structure of graph coloring is discussed first in Section 3.1, followed by a selection of typical spill heuristics in Section 3.2.

3.1 Overall Algorithm

Traditional graph coloring based register allocation was originally published in [3] and later improved in [2]. Its basic data structure is the *interference graph* $G = (V, E)$. For each VREG of a function and for each of the C available PHREGs, a node is added to V. An undirected edge $e = \{v, w\}$ is added to E whenever nodes v and w interfere. v and w interfere either if they represent VREGs which are simultaneously alive and thus should not share the same PHREG, or if a VREG v must not be allocated to PHREG w for architectural reasons.

Graph coloring assigns one of C colors representing the physical registers to each node $v \in V$ such that no two adjacent nodes have the same color. According to [2], this is done as follows:

Build: Construct the interference graph $G = (V, E)$.

Simplify: Successively remove each $v \in V$ from G having a degree $< C$, push v onto stack S. This step removes all nodes from G which are always colorable due to the small amount of adjacent nodes.

Spill: After *simplify*, each node v has degree $\geq C$. Select one node $v \in V$, mark v as *potential spill*, remove v from G, push v onto S.
If $V \neq \emptyset$, continue with *simplify*.

Select: Successively pop nodes v from S and re-insert them into G. If v is not a potential spill, assign a free color c to v. If v is a potential spill, there may be a free color c available for v. If this is the case, assign c to v. Otherwise, don't assign a color to v and mark v as *actual spill*.

Start over: If there are actual spills $v \in V$, insert a load operation before each use of v and a store operation after each definition of v and continue with *build*.

By inserting spill operations before uses and after definitions of an actual spill v during *start over*, the lifetime of v is split into smaller intervals in the hope that these smaller intervals will only interfere with lifetimes of fewer VREGs in the next iteration of the above algorithm. This register allocator has proven to produce results of high quality and has an overall complexity of $\mathcal{O}(n \log n)$, where n is the number of instructions in the program to be allocated.

3.2 Spill Heuristics

A crucial issue of the allocator sketched above is the question which node v to select and to mark as potential spill during the *spill* phase. In the related literature, several heuristics to select a potential spill are proposed:

- Select nodes according to the order in which registers occur in the compiler's intermediate code.
- Select the node v with highest degree, since spilling this node reduces the degree of many other nodes in G so that it is more likely to maximize the number of nodes with degree $< C$ after spilling v.

- Select a node v depending on the degree of v, on the number of operations o using or defining v, on the register pressure around o and on the loop nesting level of each such operation o.

From the description of these spill heuristics, it becomes obvious that no formal timing model is used to take a spilling decision. Due to a lack of such models, these heuristics try to estimate the impact of a spilling decision on code quality. It is not surprising that in some cases, one heuristic is better, whereas another heuristic might be more appropriate in other cases. Nowadays, spill decisions are not steered by actual timing data so that their estimates may guide a register allocator into a wrong direction.

4. WCET-AWARE GRAPH COLORING

The discussion of the spill heuristics in Section 3.2 reveals that current register allocators have no direct control over where spill code is generated, since only simplified measures are used. This can have severe effects on a program's WCET. The WCET of a program P is equal to the length of the longest possible execution path from the start node to an end node in P's control flow graph *(CFG)*. For such a path, its length is the sum of the products of WCET and worst-case execution frequency for all basic blocks of the path. This longest path is also known as *Worst-Case Execution Path (WCEP)*. Traditional spill heuristics may now lead to spill code generation along this WCEP, thus increasing the WCET considerably.

Since it is intractable to compute a program's WCET in general, upper bounds of the actual WCET of P need to be estimated. The state-of-the-art technique to obtain safe and tight WCET estimates is to apply static analyses to the executable machine code of P, taking into account the influence of P on memories, caches, processor pipelines etc. Static WCET analyzers like e. g. aiT [1] basically estimate *a)* the WCET per basic block of P and *b)* how many times each basic block of the CFG is executed in the worst-case (i. e. the *worst-case execution frequency* per node). Using this data, the WCEP can be determined, since any block with non-zero worst-case execution frequency belongs to the WCEP. In the remainder of this paper, the shortcuts WCET and WCET_{est} are used synonymously – we always refer to the WCET estimates produced by a static WCET analyzer in the following.

This section presents our WCET-aware graph coloring approach. Section 4.1 discusses properties of WCETs and WCEPs that heavily influence compiler optimizations aiming at WCET reduction. Section 4.2 shows the overall workflow of our WCET-aware register allocator, followed by the discussion of our WCET-aware spill heuristic in Section 4.3.

4.1 Instable Worst-Case Execution Paths

The WCET of a program P is the maximal time P's execution can ever take. The CFG of P, whose nodes represent basic blocks and whose edges tell that one basic block can be reached from the other, reflects all possible ways of executing P. Among all paths from P's start node in the CFG to some end node, there is one longest path. This path is the WCEP and its length is equal to P's WCET.

A compiler aiming at WCET minimization must thus reduce the length of the WCEP. Assume p_1 is P's current WCEP and some disjoint path p_2 is the second longest path in the CFG. If a compiler optimization is successful in shortening p_1 by more than $|p_1| - |p_2|$ time units (where $|p|$ stands for the length of p), p_2 becomes the new WCEP after this optimization.

However, if the optimization is unaware of the WCEP change from p_1 to p_2, the compiler keeps on reducing the length of p_1. Unfortunately, this effort may be in vain since it not necessarily leads to any further WCET reduction, because the new WCEP p_2 might not be affected.

As a consequence, the following requirements have to be met by compilers aiming at WCET minimization. They must

- have detailed knowledge about the WCEP,
- apply optimizations exclusively to those parts of P lying on the WCEP, since optimizing parts of P not lying on the WCEP don't reduce the WCET at all, and
- be aware of switches of the WCEP in the course of applied optimizations.

These requirements are very challenging, because they show that it is not sufficient to equip a compiler with WCET models to obtain a WCEP, but that the influence of an optimization on the model needs to be tracked steadily. A WCET-aware register allocator is even more challenging as will be shown in the next section.

4.2 A Chicken-Egg Problem

In order to design a WCET-aware register allocator, the worst-case execution frequencies per CFG node need to be known. Unfortunately, static WCET analysis can not be applied to the program P serving as input for register allocation to obtain this data. This is because P is not an executable program since it uses VREGs instead of PHREGs. Static WCET analysis relies on executable and thus register-allocated code in order to correctly take the mutual influences of P and the processor hardware into account. Hence, there are cyclic dependencies between register allocation and WCET analysis – in addition to the requirements discussed in Section 4.1 – which need to be broken in order to obtain a WCET-aware register allocator.

Conventional register allocators follow the strategy to keep as many VREGs in PHREGs as possible, and to move a VREG to memory only if this is really necessary. The traditional way of thinking thus assumes optimistically that all VREGs fit into the physical register file and that only exceptionally, a VREG is allocated to memory. This way of thinking is also reflected by the graph coloring algorithm presented in Section 3, since it first tries to remove all colorable nodes from the interference graph, and only after that, a decision on one single potential spill is taken.

However, this traditional approach is not applicable for a WCET-aware register allocator. The intermediate code produced in the course of all the steps and iterations of traditional graph coloring is not executable and thus, no WCEP can be determined. The first stage where traditional graph coloring produces executable and thus WCET analyzable code is when register allocation is just finished. And at the end of the entire procedure, it does not make sense to reason about WCEPs, since all spill decisions are already taken.

For WCET-aware graph coloring, we propose the opposite way of thinking: we assume pessimistically that all VREGs are kept in memory. During register allocation, we thus

```
1   IR WCET-GC-RA( IR $P$ ) {
2     while ( true ) {
3       IR $P'$ = $P$.copy();
4       $P'$.spillAllVREGs();

5       set<basicBlock> WCEP = computeWCEP( $P'$ );
6       if ( getVREGs( WCEP ) == $\emptyset$ )
7         break;

8       basicBlock $b'$ = getMaxSpillCodeBlock( WCEP );
9       basicBlock $b$ = getBlockOfOriginalP( $b'$ );

10      list<virtualRegister> vregs = getVREGs( $b$ );
11      vregs.sort( occurrences of VREG in $b$ );
12      traditionalGraphColoring( $P$, vregs );
13    }
14    traditionalGraphColoring( $P$, getVREGs( $P$ ) );

15    return $P$;
16  }
```

Figure 1: Algorithm for WCET-Aware Graph Coloring

move VREGs from memory to PHREGs. This approach has the advantage that the intermediate code generated in the course of register allocation is always executable so that WCEPs can be determined.

4.3 A WCET-Aware Spill Heuristic

The WCET-aware spill heuristic we present in this paper bases on the following two key characteristics:

- Due to the instability of the WCEP, it has to be recomputed regularly. Since it is practically infeasible to recompute the WCEP after each individual spill decision, we propose to recompute WCEPs after deciding on the allocation and spilling of one single basic block.
- For a given WCEP, that basic block b leading to the highest execution of spill code in the worst case is chosen for allocation. All VREGs v of b are sorted by to the number of occurrences of v in b. This precedence list of VREGs is passed to a standard graph coloring register allocator selecting that register with least occurrences in the list as potential spill, if necessary.

The overall algorithm for WCET-aware graph coloring is depicted in Figure 1. It executes an optimization loop processing one basic block per iteration (lines 2 - 13). For an *intermediate representation (IR)* of a program P being input to register allocation, the algorithm maintains a copy P' which is fully spilled, i. e. where all VREGs of P are marked as actual spills and load / store operations are inserted for spilling (lines 3 and 4). This fully spilled IR is passed to a static WCET analyzer to obtain the current WCEP, which is feasible since P' does not contain any VREGs (line 5).

Among all basic blocks on the current WCEP, that block b' with the highest execution of spill code in the worst case is selected. Worst-case execution of spill code is computed by multiplying the number of inserted spill operations per basic block with the worst-case execution frequency per basic block as determined by the WCET analyzer (line 8). For b' within the fully spilled IR P', its counterpart b within the IR P still containing VREGs is determined (line 9).

This basic block b of P is the most critical one within the current iteration of the WCET-aware register allocator. Hence, the allocator should try to keep all VREGs of b in PHREGs. However, if register pressure is too high and some VREGs of b must be spilled, this should be done such that only a minimal amount of spill code will be executed in b in the worst case. For this purpose, all VREGs v of b are sorted by their number of occurrences within b (line 11). Since spill code generation always inserts a load operation before each use of v and a store after each definition of v, the number of occurrences of v in b correlates with the amount of spill code required in b to spill v. This sorting order models a precedence which VREGs are better spill candidates and which ones not. This precedence list of VREGs is passed to a standard graph coloring allocator (cf. Section 3) which performs the actual work of mapping these virtual to physical registers (line 12). After that, b' is put in a black-list to prevent it from being selected again by line 8. For the sake of simplicity, this black-listing is omitted in Figure 1.

If the current WCEP does not contain VREGs any more, the optimization loop terminates (lines 6 and 7). However, the IR P may still contains VREGs after leaving the optimization loop. This happens for basic blocks on other paths within the CFG which have never been the WCEP – such blocks were never considered by the optimization loop. However, they still need to be allocated. For this purpose, all remaining VREGs within P are passed to a final run of the traditional graph coloring register allocator in order to obtain a fully allocated IR (line 14).

For this final run of the standard register allocator, the applied spill heuristic does not matter. This is because even in the worst case where all remaining VREGs would be spilled, the basic blocks still containing VREGs in line 14 will never ever lie on the WCEP and thus will never influence the global WCET of P. If they lay on the WCEP, they would have been captured by the optimization loop which is in contradiction to the loop's termination condition in lines 6 and 7. Hence, it is fully sufficient to provide the standard graph coloring allocator with some arbitrary precedence list of VREGs. For the sake of simplicity, we make the allocator select nodes for spilling just in the order in which VREGs occur in P.

To sum up, the algorithm sketched in Figure 1 implements a WCET-aware spill heuristic for a traditional graph coloring register allocator. It sorts VREGs using a criterion depending on worst-case execution frequencies of spill code, stemming from static WCET analysis. This way, our spill heuristic tries to avoid spill code generation along the WCEP and especially within those blocks lying on the WCEP which are most frequently executed in the worst case.

5. EVALUATION

This section presents results obtained by applying the proposed WCET-aware graph coloring allocator to real-life benchmarks. Section 5.1 describes the experimental environment used to perform benchmarking. Sections 5.2 and 5.3 discuss benchmarking results in terms of worst-case and average-case execution times, respectively.

5.1 Experimental Environment

Our WCET-aware register allocator is integrated into a compiler for the Infineon TriCore TC1796 processor. This processor features a relatively large register file having 16 data and 16 address registers. However, not all of these 32

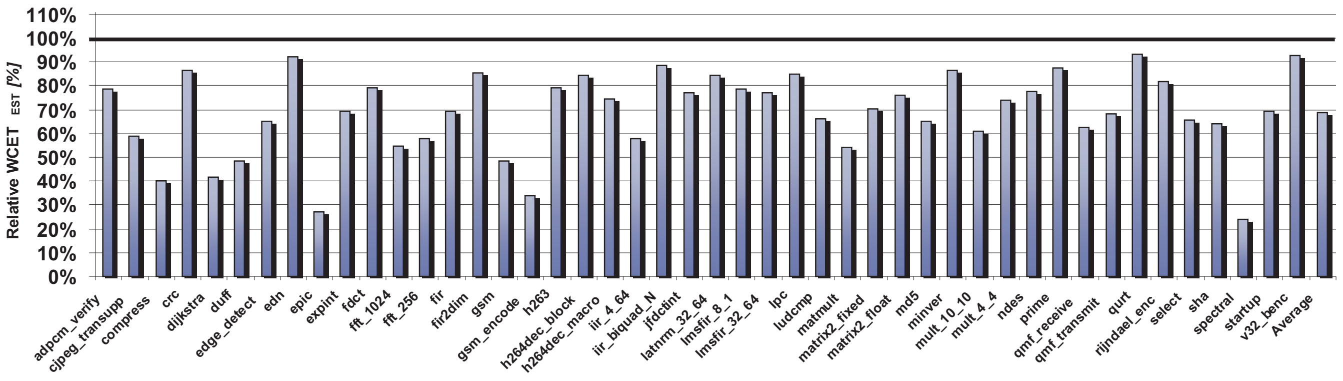

Figure 2: Relative WCET Estimates after WCET-aware Register Allocation

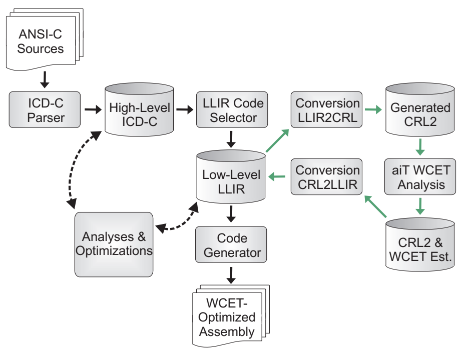

Figure 3: WCET-aware C Compiler WCC

registers can be used freely by a register allocator. Several registers are dedicated e. g. to implement function calling conventions, as stack pointer or as return address pointer.

Figure 3 depicts the structure of the WCET-aware C compiler *WCC* [5] used for benchmarking. Its key feature is the tight integration of the static WCET analyzer aiT into the compiler's backend. This way, WCET timing data is available at the compiler's assembly code representation *(ICD-LLIR)*. Both at C and at assembly level, code optimizations are applied. One of these optimizations, applied at assembly level, is the register allocation algorithm discussed in this paper. The compiler features a total of 42 different optimizations. For benchmarking, all of them are activated such that register allocation is always applied to already highly optimized code. In all experiments, spilling uses the TriCore's scratchpad memory having 1 cycle access latency.

5.2 Worst-Case Execution Time Estimates

Figure 2 shows the impact of our WCET-aware register allocation algorithm on the WCET estimates *($WCET_{est}$)* of 46 different applications from the MRTC, MediaBench, UTDSP and DSPstone benchmark suites. These benchmarks are very different: some of them are quite small filter and sorting routines, others are large and complex audio / video codecs. Their basic block counts range from 7 to 808. However, all benchmarks have in common that register pressure within their assembly code is high so that spill code needs to be generated. Figure 2 shows the $WCET_{est}$ of all benchmarks resulting from our WCET-aware register allocator as a percentage of the $WCET_{est}$ resulting from the traditional graph coloring allocator described in Section 3.

Traditional graph coloring is performed function-wise. Selecting the node with maximal number of occurrences in an entire function would make the traditional allocator optimize code size since the total amount of spill code per function is minimized. This would lead to an unfair comparison in favor of our WCET-aware register allocator. Instead, the traditional allocator spills the node with highest degree in our case. This spill heuristic tries to optimize execution time since usually, the node with highest degree interferes with other nodes in some innermost loop. Thus, this spill heuristic avoids spill code in the innermost loop which is a very similar criterion as that one used for WCET-aware register allocation, leading to a fair comparison of both allocators.

As can be seen, our WCET-aware register allocator is able to reduce the $WCET_{est}$ considerably for all benchmarks. For `qurt`, the $WCET_{est}$ after WCET-aware register allocation is 93.1% of the original $WCET_{est}$, i. e. a $WCET_{est}$ reduction of 6.9% was achieved. For all other benchmarks, even higher gains were observed. The largest gain in terms of $WCET_{est}$ was measured for `spectral` where the $WCET_{est}$ after our register allocation amounts to only 24.1% of the original $WCET_{est}$, leading to savings of 75.9%.

On average over all 46 considered benchmarks, we were able to obtain a $WCET_{est}$ of 68.8% of the original worst-case execution time estimate, corresponding to a total average $WCET_{est}$ reduction of 31.2%.

Even though our WCET-aware register allocator performs a WCET analysis for the allocation of each basic block along the WCEP, thus leading to 1,979 WCET analyses during allocation of all 46 benchmarks, the runtimes of our algorithm are still moderate. WCET-aware register allocation of all benchmarks took a total of 12:15 CPU-hours on an Intel Xeon machine running at 2.4 GHz, with an average allocation time of 16 minutes per benchmark. Of course, this is much longer than the average 10 CPU seconds per benchmark required by graph coloring, but it is still acceptable if high code quality for hard real-time systems is required.

5.3 Average-Case Execution Time & Code Size

Figure 4 shows the impact of our register allocator on average-case execution times *(ACET)* of the selected benchmarks. ACETs were measured using a commercial instruction set simulator for the TC1796. Once again, ACETs after WCET-aware register allocation are depicted as a percentage of the ACETs resulting from the traditional allocator.

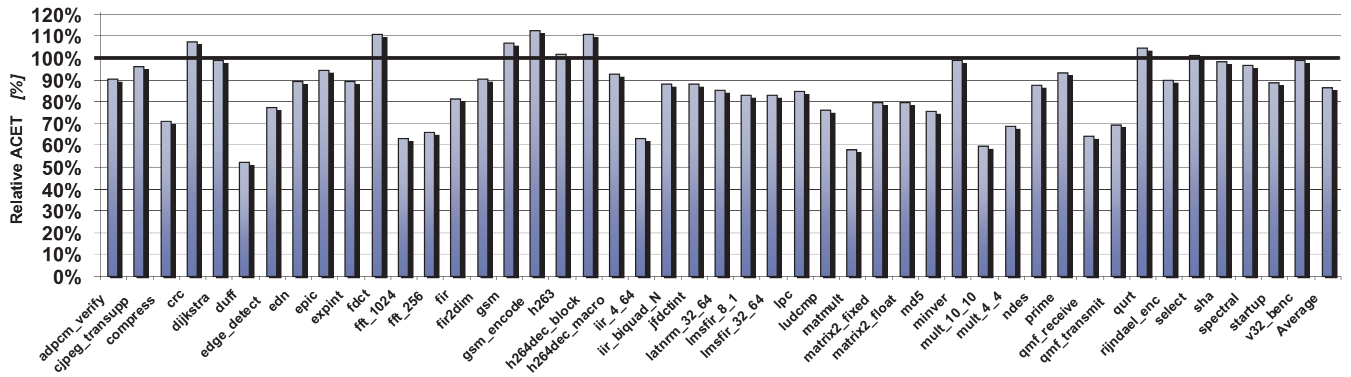

Figure 4: Relative ACETs after WCET-aware Register Allocation

A comparison of Figures 2 and 4 shows that the measured ACETs behave completely different than the WCETs resulting from our WCET-aware register allocator. For some benchmarks, our register allocator increases ACETs whereas significant WCET reductions were reported. The most prominent example is the `gsm` family of benchmarks, where WCET reductions between 51.5% – 66.2% were obtained, but ACETs degrade by 6.8% – 12.7%. In general, the WCET reductions achieved by the proposed register allocator are orders of magnitude larger than all measured ACET changes. On average for all 46 benchmarks, our register allocator leads to a total average ACET reduction of only 13.8% which is far below the obtained average WCET reduction of 31.2%.

These differences between WCETs and ACETs can be explained by the fact that our register allocator keeps on optimizing along the WCEP which is usually not identical to the path that is executed in a typical average-case scenario. Hence, our register allocator inserts spill code at positions within the CFG where it is uncritical for the worst-case performance, but may impair average-case performance. It can be concluded that optimizing for WCET is a completely different issue than optimizing for average-case performance.

For all 46 benchmarks, we observed an average increase of the benchmarks' text section of 29.8%, with a maximal increase of 298% for `dijkstra`. However, this is the only benchmark with such extreme increases. `dijkstra` is a very small benchmark kernel so that the insertion of only few additional spill instructions leads to excessive code size increases. These code size increases can be explained by the fact that our WCET-aware register allocator generates more spill code if this helps in keeping the WCEP free of spill code.

6. CONCLUSIONS

This paper is the first one to present a WCET-aware register allocator. It introduces a precise WCET model for this important compiler optimization and makes detailed WCET data available during allocation. WCET-aware register allocation is done by providing a standard graph coloring algorithm with a WCET-aware spill heuristic. The effectiveness of our approach is shown by average WCET reductions of 31.2% for 46 different real-life benchmarks. Since the proposed register allocator is explicitly tailored towards WCET minimization, it is not surprising that it reduces ACETs by only 13.8%. Code sizes increase by 29.8% on average.

Since the proposed approach to extend a traditional graph coloring allocator with a WCET-aware spilling mechanism is a heuristic trying to avoid spill code generation along the worst-case execution path, no statements can be made about the optimality of our register allocator. For this reason, our future work will concentrate on developing an integer-linear programming based WCET-aware register allocator.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using aiT (`www.absint.com/ait`).

7. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. *www.absint.com/ait*, 2009.

[2] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Apr. 1992.

[3] G. J. Chaitin, M. A. Auslander, et al. Register allocation via coloring. *Computer Languages*, 6, 1981.

[4] J.-F. Deverge and I. Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *Proceedings of ECRTS*, Pisa, July 2007.

[5] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *Proceedings of ESTIMedia*, Seoul, Oct. 2006. *ls12-www.cs.tu-dortmund.de/research/activities/wcc.*

[6] H. Falk, S. Plazar, and H. Theiling. Compile Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In *Proceedings of CODES+ISSS*, Salzburg, Oct. 2007.

[7] D. W. Goodwin and K. D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. *Software: Practice and Experience*, 26(8):929–965, Aug. 1996.

[8] E. A. Lee. Absolutely Positive On Time: What Would It Take? *IEEE Computer*, July 2005.

[9] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proceedings of ECRTS*, Prague, July 2008.

[10] G. Mandalika. Building Enterprise Applications with Sun Studio Profile Feedback. *developers.sun.com/solaris/articles/building.html*, 2007.

[11] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM TOPLAS*, 21(5):895–913, Sept. 1999.

[12] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of DATE*, Nice, Apr. 2007.

[13] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of PATMOS*, Yverdon-Les-Bains, Sept. 2001.