WCET-aware Register Allocation based on Integer-Linear Programming

Heiko Falk, Norman Schmitz, Florian Schmoll
Computer Science 12
Technische Universität Dortmund
D - 44221 Dortmund, Germany
{*Heiko.Falk, Norman.Schmitz, Florian.Schmoll*}*@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 runtime.**

This paper presents an integer-linear programming *(ILP)* based register allocator that uses precise worst-case execution time *(WCET)* models. Using this WCET timing data, the compiler avoids 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 ILP-based register allocator. Our results underline the effectiveness of the proposed techniques. For a total of 55 realistic benchmarks, we reduced WCETs by 20.2% on average and ACETs by 14%, compared to a standard graph coloring allocator. Furthermore, our ILP-based register allocator outperforms a WCET-aware graph coloring allocator by more than a factor of two for the considered benchmarks, while requiring less runtime.

***Keywords*-WCET; Register Allocation; Integer-Linear Programming; Pipeline**

I. 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 [14]. Hence, simple ad-hoc heuristics are applied during optimization in the hope that they 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.

Funded by the European Community's ArtistDesign Network of Excellence and by the European Community's 7th Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

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 WCET-aware register allocator based on formal *integer-linear programming (ILP)* models. It allows to systematically minimize a program's WCET which is desirable since highly optimized code is more cost-efficient than unoptimized code that has to be executed on faster hardware to meet its real-time constraints. The paper's main contributions are

- its ILP model of a function's WCET,
- its ILP model of a processor's pipeline in order to carefully estimate the impact of individual spill instructions on the WCET,
- the achieved WCET reductions by 20.2% and ACET reductions by 14% on average for 55 benchmarks while requiring only moderate optimization runtimes,
- the detailed comparison of the proposed ILP-based allocator with a previously published WCET-aware graph coloring allocator.

Section II gives a survey of related work on register allocation and WCET optimization. The assumed target processor is subject of Section III. Section IV presents an ILP-based register allocator minimizing spill code, followed by the proposed WCET-aware techniques in Section V. Section VI describes the benchmarking results, and Section VII summarizes this paper and gives an outlook on future work.

II. Related Work

An optimal register allocator using *integer-linear programming (ILP)* is presented in [10]. The ILP minimizes spill code overhead, i. e. the total amount of generated spill code. Since counting the number of spill instructions 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 [10] on WCETs is fully unknown. How to make this ILP WCET-aware is subject of the present paper. Due to its importance for this paper, it is discussed in more detail in Section IV.

Graph coloring is the standard technique for register allocation nowadays [2], [3]. It uses an interference graph modeling overlapping lifetimes of VREGs, and colors this graph using C colors, assuming that the considered processor has C PHREGs. Coloring is done such that no two adjacent nodes representing VREGs have the same color. A spill heuristic decides which node to place in main memory if no more free PHREGs are available.

Recently, graph coloring was made WCET-aware by introducing a dedicated spill heuristic [5]. It iteratively determines that basic block with the highest execution of spill code in the worst case and allocates all VREGs of this most critical basic block to PHREGs first. This way, the chance to insert highly undesired spill code into this most critical basic block is reduced. However, this approach still relies on a heuristic which does not consider the actual impact of spill code on a program's WCET.

Another standard register allocation technique is *linear scan* [19]. Using a linear order of all instructions 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 instruction 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 [18] as a workaround for lacking formal models. Here, information about a program's runtime behavior is collected and supplied to the compiler by applying code instrumentation plus runtime 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.

The compiler WCC [6], [7] 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 [16], a combination of procedure cloning and procedure positioning to improve worst-case I-cache performance is proposed. The authors of [4], [8], [20] 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 was considered only in [5] in the past.

III. The Target Architecture: Infineon TriCore

Throughout this paper, register allocation will be studied for Infineon TriCore [12] processors which are frequently used in the automotive domain. In order to concentrate on the novel concepts of WCET-aware ILP-based register allocation presented in the following sections, by far not all relevant technical details of the target architecture will be discussed here. Instead, we will only concentrate on those details that are necessary to understand our integer-linear programs and will omit all others for the sake of simplicity.

The TriCore's register file contains 16 data and 16 address registers. The data registers can be used freely, whereas some address registers are reserved for special purposes (e. g. for stack pointer or return address). 8 of the data and 6 of the address registers are automatically saved and restored to / from memory during function calls and returns so that they usually keep local variables. All other registers are not automatically saved and restored so that they are intended for global variables or function parameters.

Besides the ILP model of the WCET, the second key contribution of this paper is the ILP-based pipeline model that is used to estimate the influence of a spill instruction on the WCET. Thus, basic knowledge of the TriCore's pipelines is presented in the following. During register allocation, two pipelines are considered: the *I-pipeline* executes usual integer ALU instructions, whereas the *LS-pipeline* performs load / store instructions and address arithmetic. Both pipelines operate in parallel so that an I- and an LS-instruction can be executed in the same clock cycle ideally.

However, this ideal situation is defeated in the following cases:

- **Case 1:** If an I- and an LS-instruction define the same register, a classical WAW (write-after-write) hazard results:
  ```
  ADD d0,d1,d2      # d0 = d1 + d2
  LD d0,[a0]        # d0 = mem[a0]
  ```
  The TriCore solves this hazard by stalling the LS-instruction for one cycle.

- **Case 2:** If an address register is loaded from memory, it takes one additional stall cycle until the loaded address register can be used. Thus, RAW (read-after-write) hazards may occur between a load and an LS-instruction:

```
LD.A a5,[a0]      # a5 = mem[a0]
ADD.A a4,a5,a6    # a4 = a5 + a6
```

- **Case 3:** Another additional stall cycle needs to be considered if a store and a load instruction access the same memory location, due to a structural hazard between the writeback and execute stages of the LS-pipeline:

```
ST [a0],d1        # mem[a0] = d1
LD d2,[a0]        # d2 = mem[a0]
```

Due to the complexity of the TriCore pipelines, there exist several other cases where additional cycles have to be considered. As already mentioned, we omit the description of these corner cases here.

Spilling uses the stack memory that is allocated to the TriCore's *scratchpad memory (SPM)*. The SPM can be accessed without any additional wait states so that memory loads or stores generated during register allocation only take one cycle in the execute stage of the LS-pipeline.

IV. Traditional ILP-Based Register Allocation

The original publication on ILP-based register allocation [10] proposes a 0-1 ILP working function-wise, i. e. an ILP is generated and solved per function which maps virtual registers (VREGs) to physical registers (PHREGs). Binary decision variables specify to which PHREG each VREG is assigned, and when load or store instructions implementing spilling will be issued. In the following, ILP variables are represented using lowercase letters, whereas constants use uppercase letters.

Basically, the *control flow graph (CFG)* of each function is traversed. Whenever an instruction i defining a VREG v is encountered, binary decision variables $x_{i,def}^{v \to p}$ are created for all available PHREGs p. They model the situation that v is assigned to p after the execution of i. For instructions j using a VREG v, it has to be decided whether the mapping of v to p continues or ends. For this purpose, binary variables $x_{i,cont}^{v \to p}$ and $x_{i,end}^{v \to p}$ are introduced. A constraint ensures that exactly one of these two kinds of variables is set to 1. Additional variables are created for instructions i neither using nor defining v, if v is alive during the execution of i. Constraints finally ensure that the lifeness of v is propagated through such "uninvolved" instructions i.

In order to obtain a valid register allocation based on these binary decision variables, certain conditions must be fulfilled. First, the so-called *single-symbolic condition* must be met which ensures that at most one VREG can be assigned to a single PHREG at any time. For all instructions i defining a VREG v, for all PHREGs p, and for all other VREGs $w_1, w_2, \ldots$ alive at i, it must hold that at most one of v or $w_1, w_2, \ldots$ are assigned to p:

$$x_{i,def}^{v \to p} + x_{i,cont}^{w_1 \to p} + x_{i,cont}^{w_2 \to p} + \ldots \leq 1 \tag{1}$$

Second, the so-called *must-allocate condition* ensures that each VREG v is actually assigned to some PHREG p. For instructions i defining v, the following simple constraint ensures that exactly one PHREG p is used for v:

$$\sum_p x_{i,def}^{v \to p} = 1 \tag{2}$$

If instruction i uses VREG v, it is sufficient that v is contained in at least one PHREG p:

$$\sum_p (x_{i,cont}^{v \to p} + x_{i,end}^{v \to p}) \geq 1 \tag{3}$$

Up to this point, spills of registers to main memory are not yet modeled in the ILP. The authors of [10] prove that it is optimal to spill-store a VREG v to main memory either immediately after an instruction i defining v, or immediately after a basic block where control flow forks. In order to model spill-store instructions s at these positions, the ILP contains additional binary decision variables $x_{s,st}^{v \to p}$ which specify whether spill-store instruction s is created, saving VREG v originally assigned to PHREG p to main memory. Analogously, spill-load instructions s can occur either immediately before an instruction i using v, or in front of a basic block where control flow joins. Thus, decision variables $x_{s,ld}^{v \to p}$ model the decisions for such spill-loads s. Additional constraints ensure that e. g. spill-loads for a VREG v are only valid if v has been spill-stored before. Since they are not necessary to understand the remainder of this paper, these additional constraints are not shown here for the sake of clarity. Instead, the interested reader is referred to the original publication [10].

In [10], the authors propose to minimize the total spill code overhead by formulating a corresponding ILP objective function. The authors assume that each spilling-related decision of the ILP contributes by the same constant amount to the objective function. For example, under the assumption that each spill instruction has a size of 4 bytes, an objective function like e. g.

$$\sum_v \sum_p \sum_s (4 * x_{s,st}^{v \to p} + 4 * x_{s,ld}^{v \to p}) \rightsquigarrow min. \tag{4}$$

minimizes spill-related code size.

V. WCET-Aware ILP-Based Register Allocation

As stated in the previous section, the original approach for ILP-based register allocation assumes that each individual decision on spilling has the same and constant impact on the ILP's objective function. This assumption only holds for very simple kinds of objective functions like e. g. code size.

Unfortunately, it does not hold for more complex objectives like e. g. execution time, and in particular not for WCETs.

The reasons for that are twofold: First, not the entire code contributes to a function F's WCET. Only that subset of the code leading to F's worst-case behavior influences the WCET. All other parts of the code have no influence. Thus, spilling decisions of an ILP-based register allocator which are related to code portions not influencing the WCET should not have an impact on the ILP's objective function. Second, F's WCET depends on the WCETs of F's basic blocks. Spill-load and spill-store instructions generated during register allocation can have a highly variable influence on a basic block's WCET. This depends completely on a processor's hardware. Complex processor pipelines are able to execute memory accesses in parallel to arithmetic-logical instructions so that a single spill instruction possibly does not lead to an increase of a basic block's WCET in some cases. In other cases, a spill instruction might add one or more additional cycles to the block's WCET.

For these reasons, the key contributions of this paper are the consideration of only WCET-relevant code portions (cf. Section V-A), and the modeling of pipeline-related spill costs (cf. Section V-B) within the ILP formulation. The overall workflow of the proposed WCET-aware ILP-based register allocator is described in Section V-C.

A. ILP Model of the Worst-Case Execution Time

The WCET of a function F is the maximal time F's execution can ever take. The CFG of F, 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 F. Among all paths from F's start node in the CFG to some end node, there is one longest path w. r. t. execution time. This path is the *worst-case execution path (WCEP)* and its length is equal to F's WCET.

A register allocator aiming at WCET minimization must thus reduce the length of the WCEP. Assume e. g. that p_1 is F's current WCEP and some disjoint path p_2 is the second longest path in the CFG. If a register allocator successfully shortens 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 allocation. However, if the allocator is unaware of the WCEP change from p_1 to p_2, it keeps on reducing the length of p_1. 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.

Thus, a WCET-aware register allocator must have detailed knowledge about the WCEP and must be aware of changes of the WCEP in the course of the allocation. In order to capture the WCEP of a function F inside the ILP, the costs c_k per basic block b_k are modeled depending on spill decisions:

$$c_k = C_k + \sum_{s \in b_k} c_{s,spill} \tag{5}$$

Constraint (5) states that the costs of b_k are equal to the WCET C_k of b_k without any spill code, plus the WCET of spill code inside b_k. The WCET of b_k's spill code is expressed as a sum over all spill instructions s in b_k. For each such spill instruction s, its individual costs $c_{s,spill}$ are considered (cf. Section V-B).

For reducible CFGs, an innermost loop L of F has exactly one basic block b^L_{entry} being the loop's unique entry point, and possibly several back-edges turning it into a cyclic graph. Not considering these back-edges turns L's CFG into an acyclic graph. This acyclic graph without L's back-edges is denoted as $G_L = (V, E)$ in the following. The WCET w^L_{exit} of some exit node b^L_{exit} is equal to the costs of b^L_{exit}:

$$w^L_{exit} = c^L_{exit} \tag{6}$$

The WCET of a path leading from a non-exit node b_k of G_L to b^L_{exit} must be greater than or equal to the WCET of any successor of b_k in G_L, plus the costs b_k causes. Thus, if b_k has exactly one successor b_{succ}, the following constraint is generated:

$$\forall b_k \in \{V \setminus b^L_{exit} \mid \exists_{=1}(b_k, b_{succ}) \in E\} : w_k = w_{succ} + c_k \tag{7}$$

If there are $x > 1$ successors for a node b_k, x inequalities of the following kind are created:

$$\forall b_k \in V \setminus \{b^L_{exit}\} : \forall (b_k, b_{succ}) \in E : w_k \geq w_{succ} + c_k \tag{8}$$

Variable w^L_{entry} models the WCET of all paths of loop L if it is executed exactly once. To model several executions of L, all CFG nodes $v \in V$ of G_L are merged to a new super-node v_L. The costs of v_L are the product of L's WCET if executed once and L's maximal loop iteration count:

$$c_L = w^L_{entry} * I^L_{max} \tag{9}$$

Replacing a loop L by a super-node v_L in the CFG may turn another loop L' of F directly surrounding L into an innermost loop with acyclic CFG G'_L. Hence, the constraints of Equations (6), (7) and (8) can be formulated for L'. This way, the innermost loops of F are successively collapsed in the CFG so that ILP constraints modeling F's control flow are created from the innermost to the outermost loops.

A program's WCEP can change during optimization only at such points in the CFG where a basic block b_k has more than one successor because only there, different paths in the control flow start and the longest one of them being the actual WCEP needs to be considered. Since constraint (8) is formulated for each successor of block b_k, variable w_k always reflects the WCET of any path starting from b_k – irrespective of the fact which of the successors actually lies on the current WCEP. This way, constraint (8) realizes the implicit consideration of WCEPs and their changes in the ILP.

The WCET of a function F is the maximal execution time of a path starting at F's entry node b^F_{entry}. Thus, the

WCET of F is modeled by variable w^F_{entry}. Since ILP-based register allocation is applied function-wise (cf. Section IV), the objective function of the proposed WCET-aware ILP for register allocation is to minimize the WCET of a function F. Since F's WCET is modeled using variable w^F_{entry}, the value of this decision variable simply needs to be minimized by the ILP:

$$w^F_{entry} \rightsquigarrow min. \tag{10}$$

B. ILP Model of Pipeline-Related Spill Costs

Equation (5) uses variables $c_{s,spill}$ to model the WCET of a potentially inserted spill instruction s. However, the WCET of s depends on several factors: on s itself, and on the instructions surrounding s.

If s is inserted immediately after some I-instruction i, and if no WAW conflict between s and i exists, s is executed in parallel and thus costs 0 cycles. If a WAW dependence between s and i exists, s has costs of 1 cycle (cf. case 1 in Section III). Unfortunately, inserting a spill instruction s might also break some previously existing parallelism:

```
ADD d0,d1,d2        # i: d0 = d1 + d2
LD d3,[a0]          # s: d3 = mem[a0]
ST [a5],d3          # j: mem[a5] = d3
```

Before the insertion of s in this example, i and j are executed in parallel. Afterwards, s is executed in parallel to i so that the costs of s are zero. However, j is no longer executed in parallel and now costs 1 cycle which is caused by the insertion of spill instruction s.

These effects caused by the insertion of a spill instruction s – namely the costs of s itself, and the costs caused by s due to lost parallelism – are called *direct costs*. In the ILP, integer variables $c_{s,direct}$ model the direct costs of spill s.

Cases 2 and 3 in Section III consider additional costs due to extra pipeline stalls caused by RAW and structural hazards. These so-called *stall costs* of spill instruction s are represented by ILP variables $c_{s,stall}$.

Using these variables for direct and stall costs, the overall costs of a spill instruction s are:

$$c_{s,spill} = c_{s,direct} + c_{s,stall} \tag{11}$$

The following subsections describe how to compute these direct and stall costs in the ILP.

Direct Spill Costs

In order to determine direct spill costs for a potential spill instruction s spill-loading or spill-storing VREG v, the cases specified below need to be checked, in the given order. If a case matches the situation for spill s, the constraints given for the particular case are inserted into the ILP, and the remaining cases are not checked any more.

1) If the instruction i immediately before s is not an I-instruction, or if i is an I-instruction, but a definite WAW conflict between s and i exists, s can not be executed in parallel to i. In this case, the direct spill costs of s are equal to 1, i. e. equal to the decision whether spill instruction s has to be inserted or not:

$$c_{s,direct} = \begin{cases} \bigvee\limits_p x^{v\to p}_{s,st} & \text{if } s \text{ is spill-store} \\ \bigvee\limits_p x^{v\to p}_{s,ld} & \text{if } s \text{ is spill-load} \end{cases} \tag{12}$$

2) If s is a spill-load and if the immediate predecessor i of s is an I-instruction, the direct costs of s depend on whether s has to be inserted, and whether a WAW conflict between s and i possibly exists:

$$c_{s,direct} = \bigvee_p (x^{v\to p}_{s,ld} \wedge x^{i,p}_{s,WAW}) \tag{13}$$

The binary decision variable $x^{i,p}_{s,WAW}$ states whether spill instruction s defines PHREG p and instruction i also defines PHREG p. Thus, it models the presence of a WAW conflict between s and i via p.

3) The above two cases assume that a potential spill s is inserted immediately after some other original instruction i. However, it can happen that several different spill instructions $s_1, \ldots, s_n$ may be inserted in a sequence after an instruction i:

```
ADD d0,d1,d2        # i
LD ...              # s_1
...
ST ...              # s_m
...
LD ...              # s_n
```

Whether or not spill instructions s_1 to s_n will be generated depends on decision variables $x^{v\to p}_{s_j,st}$ and $x^{v\to p}_{s_j,ld}$, $1 \le j \le n$, resp., as mentioned previously. Such a sequence of possible spill instructions is called a *spill block*. The costs of a spill instruction s_m somewhere in the middle of such a spill block now additionally depend on the information whether any of its preceding spill instructions $s_1, \ldots, s_{m-1}$ in the spill block will be generated. If at least one of these preceding spill instructions will be generated, s_m can not be executed in parallel and causes one cycle of direct costs. The generation of at least one spill instruction preceding s_m in a spill block is encoded in the binary variable $x_{s_m,pred}$.

Thus, the direct costs of a spill instruction s_m, $1 < m \le n$, somewhere inside a spill block are the straightforward extension of Equations (12) and (13) by this novel variable $x_{s_m,pred}$:

$$c_{s_m,direct} = \begin{cases} c^{store}_{s_m,direct} & \text{if } s_m \text{ is spill-store} \\ c^{load}_{s_m,direct} & \text{if } s_m \text{ is spill-load} \end{cases} \tag{14}$$

$$c^{store}_{s_m,direct} = (\bigvee_p x^{v\to p}_{s_m,st}) \wedge x_{s_m,pred} \tag{15}$$

$$c^{load}_{s_m,direct} = (\bigvee_p x^{v\to p}_{s_m,ld}) \wedge (\bigvee_p x^{i,p}_{s_m,WAW} \vee x_{s_m,pred}) \tag{16}$$

The above cases 1) – 3) model the costs of a spill instruction s itself. Similar to the above equations, constraints are produced which model the relationship between a spill instruction s and a following LS-instruction j where previously existing parallelism gets lost. Since these constraints do not provide additional insight into our pipeline-related spill cost model, we do not list these constraints here.

Equations (13) and (16) depend on the detection of WAW conflicts via variables $x^{i,p}_{s,WAW}$. In order to determine such WAW conflicts, it has to be verified whether spill instruction s defines the same PHREG p as instruction i. For this purpose, it is sufficient to examine all VREGs v defined or spill-loaded by both s and i, and to check that no two of these VREGs are assigned to the same PHREG p:

$$x^{i,p}_{s,WAW} = \Big(\bigvee_{v\in s}(x^{v\to p}_{s,def} \vee x^{v\to p}_{s,ld})\Big) \wedge \Big(\bigvee_{v\in i}(x^{v\to p}_{i,def} \vee x^{v\to p}_{i,ld})\Big) \tag{17}$$

Furthermore, Equations (15) and (16) require information $x_{s_m,pred}$ whether some preceding spill instructions in a spill block are generated. For this purpose, variable $x_{s_k,spill}$ states whether a spill instruction s_k is generated at all:

$$x_{s_k,spill} = \begin{cases} \bigvee_p x^{v\to p}_{s_k,st} & \text{if } s_k \text{ is spill-store} \\ \bigvee_p x^{v\to p}_{s_k,ld} & \text{if } s_k \text{ is spill-load} \end{cases} \tag{18}$$

$x_{s_k,spill}$ is now used for all preceding instructions in a spill block to determine $x_{s_m,pred}$:

$$x_{s_m,pred} = \bigvee_{k=1}^{m-1} x_{s_k,spill} \tag{19}$$

In the above equations, the operators $\vee$ and $\wedge$ represent the Boolean OR and AND of two binary decision variables, resp. These Boolean operators can be modeled within the ILP, but we omitted the listing of these constraints for the sake of simplicity.

Stall Costs

The stall costs of a spill instruction s_m somewhere inside a spill block are the additional cycles s_m is stalled due to cases 2 and 3 in Section III. Thus, the stall costs $c_{s_m,stall}$ are the sum of the cycles for these two individual cases:

$$c_{s_m,stall} = c_{s_m,case\ 2} + c_{s_m,case\ 3} \tag{20}$$

For case 2, we need to distinguish whether s_m is a spill-store or a spill-load:

$$c_{s_m,case\ 2} = \begin{cases} c^{store}_{s_m,case\ 2} & \text{if } s_m \text{ is spill-store} \\ c^{load}_{s_m,case\ 2} & \text{if } s_m \text{ is spill-load} \end{cases} \tag{21}$$

For a spill-load, it has to be checked whether s_m defines a PHREG p used by some following LS-instruction j performing e. g. address arithmetic based on p. In other words, an additional stall cycle is counted if spill-load s_m is actually generated and an RAW conflict between s_m and j exists, and if successive spill instructions $s_{m+1},\ldots,s_n$ in the spill block are not generated. These three conditions are expressed in a straightforward way as follows:

$$c^{load}_{s_m,case\ 2} = \Big(\bigvee_p (x^{v\to p}_{s_m,ld} \wedge x^{j,p}_{s_m,RAW})\Big) \wedge \overline{x_{s_m,succ}} \tag{22}$$

If, in contrast, s_m is a spill-store, it has to be checked whether s_m uses a PHREG p which is defined by some preceding load instruction i, and that spill instructions $s_1,\ldots,s_{m-1}$ of a spill block are not created:

$$c^{store}_{s_m,case\ 2} = \Big(\bigvee_p (x^{v\to p}_{s_m,st} \wedge x^{i,p}_{s_m,RAW})\Big) \wedge \overline{x_{s_m,pred}} \tag{23}$$

Stall case 3 of Section III models the situation that a store accesses the same memory location as an immediately following load instruction. Since the ILP only considers pipeline-related costs of spill instructions, it is sufficient to only consider spill-stores and spill-loads here; general stores / loads performing data accesses do not need to be modeled here. Thus, the following situation potentially leads to an additional stall cycle for a spill-load s_m:

```
ST [a0],...        # s_s: mem[a0] = ...
...                # some spill block after s_s
I                  # i : an arbitrary I-instruction
...                # some spill block in front of s_m
LD ...,[a0]        # s_m: ... = mem[a0]
```

A spill-store s_s, to which some arbitrary I-instruction i is executed in parallel, accesses the same memory location as a following spill-load s_m. In this situation, s_m costs one additional stall cycle, if both s_s and s_m are actually generated and if no other spill instructions are inserted after s_s and before s_m.

Since spill-stores and -loads only access memory locations on the stack, and since it can be assumed without loss of generality that each VREG v obtains its unique place on the stack if spilled, the test whether both s_s and s_m access the same memory location is equivalent to the question whether s_s and s_m both store and load the same VREG v. Since this question can be answered off-line outside the ILP by simply checking which VREG s_s and s_m actually spill, this test does not need to be explicitly formulated in the ILP. Instead, it is sufficient to model the additional stall costs for case 3 of Section III if and only if s_m and s_s spill the same VREG.

Putting all these considerations together yields the following constraint:

$$c_{s_m,case\ 3} = x_{s_s,spill} \wedge x_{s_m,spill} \wedge \overline{x_{s_s,succ}} \wedge \overline{x_{s_m,pred}} \tag{24}$$

Similar to the direct spill costs described previously, Equations (22) to (24) rely on the detection of RAW conflicts $x^{i,p}_{s_m,RAW}$ between two instructions s_m and i, and

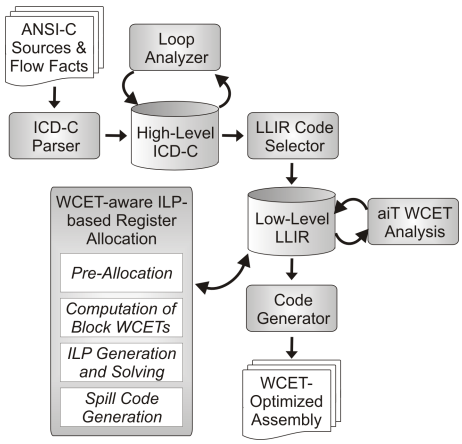

Figure 1. WCET-aware C Compiler WCC

on the information $x_{s_m,succ}$ whether some successive spill instructions in a spill block are generated. Constraints analog to Equations (17) and (19) compute these variables.

C. Workflow of WCET-aware ILP-based Register Allocation

To turn the ILP model presented in Sections V-A and V-B into a fully functional optimization, support by an underlying compiler infrastructure is required. In particular, we employ the infrastructure of our WCET-aware C compiler *WCC* [6] for the Infineon TriCore TC1796 and TC1797 processors (cf. Figure 1) to extract all the constants required by the ILP from the code currently under optimization.

Equation (9) depends on a loop's maximal iteration count I^L_{max}. In our compiler, this value can stem from user-specified flow fact annotations, or it can be generated by our automatic loop analyzer [15]. Irrespective of the origin of I^L_{max}, flow fact mechanisms take care to keep the values I^L_{max} up to date during all loop optimizations of our compiler such that correct values are used by our ILP for register allocation.

Equation (5) depends on the constant C_k representing the WCET of basic block b_k without any spill code. The determination of these constants requires massive support by our compiler infrastructure. A static WCET analyzer like e. g. aiT [1] provides the desired information about a block's WCET. Unfortunately, static WCET analysis can not be applied to the program P serving as input for register allocation. 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 between P and the processor hardware into account.

As a consequence, VREGs need to be replaced by PHREGs in order to be able to apply static WCET analysis and in order to finally obtain the required constants C_k. In other words, a register allocation mandatorily needs to be performed prior to solving the WCET-aware ILP proposed in Section V. This mandatory register allocation is called *pre-allocation* (cf. Figure 1). For pre-allocation, any existing kind of register allocator can be used. Due to its usually high quality and very fast runtimes, we use a register allocator based on graph coloring [2] for pre-allocation.

Internally, our WCET-aware ILP-based register allocator generates a copy P' of the program P which is to be register allocated. Graph coloring is applied to P' during pre-allocation in a very first step.

After pre-allocation, aiT is applied to P' to determine the individual WCETs per basic block b_k. However, it is likely that the pre-allocation algorithm needs to insert spill code. Thus, there may exist basic blocks in P' with spill code so that aiT provides block WCETs including this spill code overhead. For this reason, the block WCETs computed by aiT for program P' need to be analyzed and the block WCETs for P without spill code have to be computed in a second step.

For this purpose, the basic blocks $b'_k \in P'$ and the possibly included spill instructions are examined. Based on the characteristics of the TriCore's pipeline already described in Section III, it is determined whether a spill instruction $s \in b'_k$ is actually executed in parallel to some other instruction, and whether s causes additional stall cycles. The WCET of block b'_k is reduced by the so-computed number of cycles per spill instruction s, yielding the desired WCET C_k of the original block $b_k \in P$ without any spill code. After this step, the temporary copy P' of P is removed since it is no longer used.

Based on these block WCETs and maximal loop iteration counts, the ILP presented in Section V is generated and solved using the ILP solver *cplex* in a third step.

After solving the ILP, the values for the decision variables $x^{v\to p}_{i,def}$, $x^{v\to p}_{i,cont}$ and $x^{v\to p}_{i,end}$ specify to which PHREG p each VREG v is allocated. Furthermore, variables $x^{v\to p}_{s,st}$ and $x^{v\to p}_{s,ld}$ state whether spill instructions s have to be inserted. Based on this information, the code of the program P currently under optimization is transformed in a fourth and final step such that it reflects exactly the allocation decisions taken by the ILP.

VI. Evaluation

This section presents results obtained by applying the proposed WCET-aware ILP-based register allocator to real-life benchmarks. Section VI-A describes the experimental environment used to perform benchmarking. Sections VI-B and VI-C discuss benchmarking results in terms of worst-case and average-case execution times, respectively. Finally, the allocator's runtimes are subject of Section VI-D.

A. Experimental Environment

Our WCET-aware register allocator is fully integrated into the WCC compiler (cf. Figure 1). Its key feature is the tight integration of the static WCET analyzer aiT into the

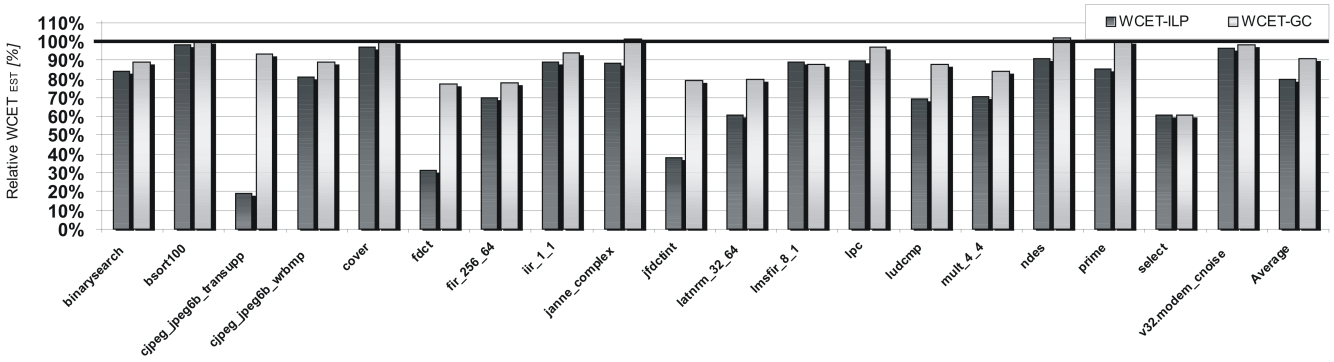

Figure 2. Relative WCETs after WCET-aware Register Allocation

compiler's backend. This way, WCET timing data is available at the compiler's assembly code representation *(ICD-LLIR)*. Both at C and assembly level, code optimizations are applied. One of these optimizations, applied at assembly level, is the register allocation technique described in this paper. The compiler features a total of 42 different optimizations. For benchmarking, all of them are activated using optimization level *-O3* such that register allocation is always applied to already highly optimized code. Furthermore, the TC1796 processor was considered during all experiments.

Our ILP-based register allocator was applied to a total of 55 different real-life benchmarks from the MRTC [17], MediaBench [13], MiBench [11], UTDSP [22] benchmark suites and from a few other sources. The benchmarks are very different: some of them are quite small filter or sorting routines, others are large and complex audio / video codecs. Their basic block counts range from 4 to 422, their numbers of virtual registers vary from 30 to 994. However, all benchmarks have in common that register pressure is high so that spill code needs to be generated.

In the following sections, the impact of our WCET-aware ILP-based register allocator on both worst- and average-case execution times will be discussed. In order to determine these results, the assembly code generated after register allocation is fed into aiT for a final analysis yielding WCET results on the one hand. On the other hand, the same assembly code is simulated using the cycle-true instruction set simulator *CoMET* [21] in order to obtain *average-case execution times (ACET)*.

B. Worst-Case Execution Time Estimates

Figure 2 shows the impact of our WCET-aware ILP-based register allocator on the WCET estimates *($WCET_{est}$)* of our benchmarks. For the sake of readability, only a subset of all 55 benchmarks is depicted. The figure shows the $WCET_{est}$ after our WCET-aware ILP-based register allocator as a percentage of the $WCET_{est}$ resulting from WCC's optimization level *-O3* and traditional graph coloring allocation [2] (bars "WCET-ILP").

As can be seen, our WCET-aware ILP-based register allocator is able to reduce the $WCET_{est}$ considerably. For a few benchmarks like e. g. `bsort100`, only marginal $WCET_{est}$ reductions by $1\% - 2\%$ were observed. For all other benchmarks, significantly higher gains were achieved. The largest gain in terms of $WCET_{est}$ was measured for `cjpeg_jpeg6b_transupp` where the $WCET_{est}$ after our ILP-based register allocation amounts to only 19.2% of the original $WCET_{est}$, leading to savings of 80.8%.

On average over all 55 considered benchmarks, we were able to obtain a $WCET_{est}$ of 79.8% of the original worst-case execution time estimate, corresponding to a total average $WCET_{est}$ reduction of 20.2%.

In addition to the above results, Figure 2 also includes the comparison of our proposed ILP-based register allocator with the WCET-aware graph coloring approach from [5] (bars "WCET-GC"). It can be seen that the WCET-aware graph coloring sometimes is unable to improve over the WCET-unaware register allocator denoted by the 100% baseline, whereas our WCET-aware ILP-based allocator achieves significant $WCET_{est}$ reductions (e. g. for `janne_complex`, `ndes` or `prime`). On average over all 55 benchmarks, the WCET-aware graph coloring heuristic achieves $WCET_{est}$ reductions by 9%.

This result shows that, for the used benchmark set, our ILP-based approach achieves $WCET_{est}$ reductions which are more than a factor of 2 larger than those achieved by WCET-aware graph coloring. This result clearly demonstrates the power of our proposed integer-linear program. Furthermore, it underlines that it is worthwhile to spend some effort in a careful and precise formal modeling of a program's execution characteristics like e. g. WCET and pipeline effects.

C. Average-Case Execution Times

Figure 3 shows the impact of our register allocator on our benchmarks' ACETs. Once again, ACETs after WCET-aware register allocation are depicted as a percentage of

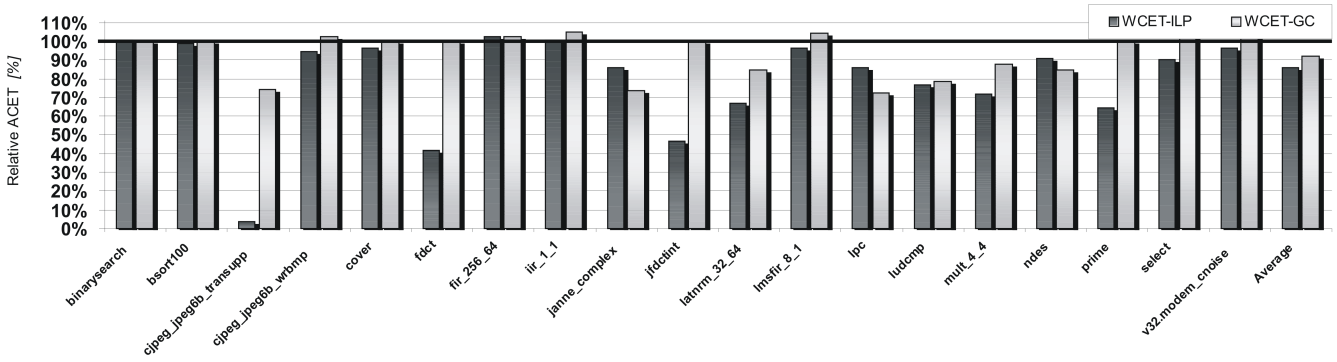

Figure 3. Relative ACETs after WCET-aware Register Allocation

the ACETs resulting from optimization level *-O3* and the traditional WCET-unaware register allocator.

A comparison of Figures 2 and 3 shows that the measured ACETs behave completely different than the WCET_{est} resulting from our WCET-aware register allocator. For some benchmarks, our register allocator increases ACETs whereas significant WCET_{est} reductions were reported. One example is the `fir_256_64` benchmark, where a WCET_{est} reduction of 30.2% was obtained by our ILP-based register allocator, but ACET increases by 2.2%. In general, the WCET_{est} reductions achieved by the proposed register allocator are usually significantly larger than the measured ACET changes. On average for all 55 benchmarks, our ILP-based register allocator leads to a total average ACET reduction of 14% which is less than the obtained average WCET_{est} reduction of 20.2%.

These differences between WCET_{est} and ACET conform to the observations in [5]. They can be explained by the fact that our WCET-aware 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.

The direct comparison between our novel ILP-based register allocator and the WCET-aware graph coloring heuristic in Figure 3 shows that even w. r. t. ACET, our ILP-based approach is better than WCET-aware graph coloring on average. While WCET-aware graph coloring leads to average ACET reductions of 8.3%, our ILP-based WCET-aware allocator reduces ACETs by 14% on average over all 55 benchmarks. However, some benchmarks like e. g. `janne_complex`, `lpc` or `ndes` show that WCET-aware graph coloring may reduce ACET stronger than the ILP-based allocator.

This behavior can again be explained by the allocators' focus on WCET. The graph coloring allocator of [5] applies a relatively simple heuristic deciding which VREG to spill into memory which does not take the actual impact of individual spill instructions on a basic block's WCET into account. This way, this allocator tries to keep the WCEP free of spill code. However, it may not always succeed in doing so. For the three benchmarks `janne_complex`, `lpc` or `ndes`, Figure 2 reveals that graph coloring does not reduce WCET_{est}. Thus, it can be concluded that WCET-aware graph coloring sometimes inserts spill code at sub-optimal positions in the benchmarks' codes which are beneficial for the ACET instead of the WCET by accident.

In contrast, our novel WCET-aware ILP-based register allocator takes the impact of each individual spill instruction into account during optimization. Thus, it is able to reduce WCET_{est} in a much more systematic way than graph coloring. However, it may still happen that our ILP-based allocator reduces ACET more than WCET_{est} (e. g. for `cjpeg_jpeg6b_transupp`, `janne_complex`, `lpc` or `prime`). This observation is again caused by the different nature of worst- and average-case execution paths *(ACEP)*. By definition, the WCEP is longer than a path taken in the average case. If both WCEP and ACEP share some common basic blocks B, our ILP-based register allocator might reduce the execution time of B by a certain amount c of cycles. This reduction translates to a shortening of the WCEP and thus to a reduction of the WCET_{est} by e. g. $x\%$. However, the same reduction c of cycles translates to a shortening of the ACEP by, say, $y\%$ with $y > x$ since the ACEP is shorter than the WCEP so that the same number c of saved cycles shortens the ACEP by a larger percental amount.

D. Runtimes

For the considered 55 benchmarks, the overall runtimes on a 2.4 GHz PC – including the CPU time required by the pre-allocation, WCET analyzer and ILP solver – are very moderate. They range from 1 CPU second for benchmark `iir_1_1` up to a maximal runtime of 54:08 CPU minutes for benchmark `cjpeg_jpeg6b_transupp`. On average

over all 55 benchmarks, our ILP-based register allocator only takes 03:33 CPU minutes per benchmark.

In contrast, the WCET-aware graph coloring allocator requires 04:13 CPU minutes on average per benchmark. This higher runtime of graph coloring is caused by the fact that it performs a costly WCET analysis after register allocation for each individual basic block.

Thus, it can be concluded that our ILP-based allocator provides high-quality code while requiring less runtime compared to WCET-aware graph coloring.

VII. Conclusions

This paper is the first one to present a WCET-aware ILP-based register allocator. It introduces a sophisticated WCET and pipeline model integrated into an integer-linear program. This way, the proposed register allocator can take the impact of potentially inserted individual spill instructions on the WCET into account in a very precise fashion. The effectiveness of our approach is shown by average WCET_{est} reductions of 20.2% for 55 different real-life benchmarks. In addition to these WCET_{est} reductions, the proposed register allocator reduces ACETs by 14% on average even though ACET is not subject of our model. Compared to a previously published work on WCET-aware register allocation, our ILP-based approach is able to outperform WCET-aware graph coloring by more than a factor of 2 in terms of WCET_{est}.

In the future, we plan to improve the runtimes of our ILP-based allocator by simplifying the ILP such that redundant spill decisions are no longer modeled [9]. Additionally, the quality of the resulting code in terms of WCET could be further improved by integrating rematerialization into our ILP formulation.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the aiT framework. The authors would also like to thank Synopsys for the provision of the instruction set simulator CoMET enabling the determination of ACETs.

References

[1] AbsInt Angewandte Informatik GmbH, "aiT: Worst-Case Execution Time Analyzers," http://www.absint.com/ait, Apr. 2011.

[2] P. Briggs, "Register Allocation via Graph Coloring," Ph.D. dissertation, Rice University, Houston, Apr. 1992.

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

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

[5] H. Falk, "WCET-aware Register Allocation based on Graph Coloring," in *Proceedings of DAC*, San Francisco, Jul. 2009.

[6] H. Falk and P. Lokuciejewski, "A compiler framework for the reduction of worst-case execution times," *Springer Real-Time Systems*, vol. 46, no. 2, Oct. 2010.

[7] H. Falk, P. Lokuciejewski, and H. Theiling, "Design of a WCET-Aware C Compiler," in *Proceedings of ESTIMedia*, Seoul, Oct. 2006.

[8] 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.

[9] C. Fu and K. D. Wilken, "A faster optimal register allocator," in *Proceedings of MICRO*, Istanbul, Nov. 2002.

[10] D. W. Goodwin and K. D. Wilken, "Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming," *Software - Practice and Experience*, vol. 26, no. 8, Aug. 1996.

[11] M. R. Guthaus, J. S. Ringenberg, D. J. Ernst *et al.*, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of WWC*, Austin, Dec. 2001.

[12] *TriCore 1 Architecture Overview Handbook*. Infineon Technologies AG, Document Revision V1.3.3, May 2002.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of MICRO*, Washington DC, Dec. 1997.

[14] E. A. Lee, "Absolutely Positively On Time: What Would It Take?" *Embedded Systems Column, IEEE Computer*, Jul. 2005.

[15] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models," in *Proceedings of CGO*, Seattle, Mar. 2009.

[16] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven Cache-based Procedure Positioning Optimizations," in *Proceedings of ECRTS*, Prague, Jul. 2008.

[17] Mälardalen WCET Research Group, "WCET Benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, Apr. 2011.

[18] G. Mandalika, "Building Enterprise Applications with Sun Studio Profile Feedback," http://developers.sun.com/solaris/articles/building.html, Jul. 2007.

[19] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, Sep. 1999.

[20] 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.

[21] Synopsys, Inc., http://www.synopsys.com, Apr. 2011.

[22] "UTDSP Benchmark Suite," www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html, Apr. 2011.