Optimal Static WCET-aware Scratchpad Allocation of Program Code*

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

Jan C. Kleinsorge
Computer Science 12
Technische Universität Dortmund
D - 44221 Dortmund, Germany
Jan.Kleinsorge@tu-dortmund.de

ABSTRACT

Caches are notorious for their unpredictability. It is difficult or even impossible to predict if a memory access will result in a definite cache hit or miss. This unpredictability is highly undesired especially when designing real-time systems where the *worst-case execution time (WCET)* is one of the key metrics. *Scratchpad memories (SPMs)* have proven to be a fully predictable alternative to caches. In contrast to caches, however, SPMs require dedicated compiler support.

This paper presents an optimal static SPM allocation algorithm for program code. It minimizes WCETs by placing the most beneficial parts of a program's code in an SPM. Our results underline the effectiveness of the proposed techniques. For a total of 73 realistic benchmarks, we reduced WCETs on average by 7.4% up to 40%. Additionally, the run times of our ILP-based SPM allocator are negligible.

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, Scratchpad 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.

For hard real-time systems, caches are problematic. Since they are hardware controlled, it is virtually impossible to determine the latency of a memory access for many popular cache architectures. Additionally, sporadically executed code like e. g. scheduler or interrupt handlers may modify cache contents. Hence, statically determined WCET estimates may be heavily overestimated in the presence of caches. Therefore, real-time system designers tend to disable caches. Such systems suffer a low average-case performance since each memory access uses the slow main memory.

Scratchpad memories (SPMs) have both a good average and worst-case performance. This paper presents a WCET-aware SPM allocation of program code. Our algorithm determines a static SPM allocation: the SPM's contents is pre-computed at compile time and remains fixed during run time. Due to *integer-linear programming (ILP)*, our approach is optimal in that it results in a minimal WCET.

A program P's WCET is the maximal time P's execution can ever take. The *control flow graph (CFG)* of P, whose nodes represent basic blocks and whose edges indicate 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, called *worst-case execution path (WCEP)*, and its length is equal to P's WCET. Here, path length is the sum of the products of WCET and worst-case execution frequency for all basic blocks of the path.

A WCET minimizing compiler thus has to reduce the WCEP's length. Assume e. g. that p_1 is P's current WCEP and some disjoint path p_2 is the second longest path in the CFG. If an optimization 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 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.

Hence, 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 ly-

*Funded by the European Community's 7th Framework Programme FP7/2007-2013 under grant agreement n° 216008.

DAC'09, July 26-31, 2009, San Francisco, California, USA

ing on the WCEP, since optimizing parts of P not lying on the WCEP don't reduce the WCET at all, and
- be aware of changes of the WCEP in the course of applied optimizations.

These requirements are very challenging, because such instable WCEPs are difficult to consider within compiler optimizations. This paper is the first one to present an optimal WCET-aware SPM allocation technique for program code. The main contributions of the proposed approach are that it

- inherently captures a program's current WCEP and its possible switches,
- improves the current state of the art of SPM allocation of program code. Our technique is the first one to consider jump penalties and variable basic block sizes based on jump scenarios.
- achieves average WCET reductions from 7.4% up to 40% for 73 real-life benchmarks while requiring only negligible optimization run times.

Section 2 gives a survey of related work. Section 3 presents our ILP model to perform optimal WCET-aware SPM allocation of program code. The link between our ILP model and a compiler infrastructure used to extract all constants required by the ILP is described in Section 4. Section 5 describes the benchmarking results, and Section 6 gives a summary and an outlook on our future work.

2. RELATED WORK

Compiler-guided SPM allocation has been studied intensely in the past. SPMs are frequently used to reduce average-case performance or energy dissipation. Generally, global data, basic blocks, sequences of basic blocks or functions are placed in an SPM to realize a certain profit in terms of run time or energy dissipation. As pointed out in [14, 18], simply greedily selecting the SPM contents can lead to suboptimal or even degraded results. Among all proposed allocation strategies for energy dissipation or run time minimization, ILP-based approaches are most popular due to the optimality of the results and the elegance of the models.

An ILP for static SPM allocation of functions, basic blocks and data minimizing energy dissipation is proposed in [11]. It introduces multi basic blocks to model the fact that sizes and energy savings of basic blocks vary depending on a current SPM allocation due to the insertion of additional jump instructions. This concept basically fully enumerates the power set of all basic blocks in the ILP, thus leading to an exponential explosion of the ILP's size. This exponential ILP explosion is avoided in the present paper by modeling variable block sizes and gains using so-called jump scenarios.

In [16, 17], the impact of SPMs on WCET prediction is studied. Even though WCET is subject of these papers, the ILP-based SPM allocation is not WCET-aware. Instead, an energy minimizing selection algorithm is employed, and the effect of this energy reduction strategy on WCET is evaluated afterwards. Hence, that work is not a true WCET-aware optimization and does not consider WCEPs at all.

Software controlled caches allowing to load contents into a cache and to lock it afterwards, i. e. to prevent it from being replaced, behave like SPMs. All following publications explicitly focus on WCET minimization. In [2], a genetic algorithm for cache contents selection of statically locked I-caches is presented. However, this approach does not necessarily yield optimal results. An explicit search for the WCEP within the CFG is performed in [6] and I-cache contents selection is done along the found WCEP. [6] relies on repeatedly investigating the CFG and is therefore expensive to perform. A similar iterative approach was presented in [10]: multiple optimization steps along the current WCEP are performed without recomputing the WCEP. After a certain number of optimization steps, the partially optimized program is analyzed and the resulting WCEP is computed for subsequent optimization steps.

The authors of [3] present a hybrid approach for WCET-centric dynamic SPM allocation of data. It is hybrid in that sense that it combines an ILP with an iterative heuristic. Using a static WCET analyzer, the current WCEP is computed. After that, an ILP tailored for this particular WCEP is solved that determines optimally which data is allocated to the SPM. In the next iteration, the WCEP is recomputed and some more SPM contents is determined using ILP.

In [12], a fully ILP-based solution to the problem of static allocation of data to SPMs for WCET reduction is presented. This work serves as basis for the techniques presented in the following. However, [12] is unable to allocate code onto SPMs and suffers from several limitations preventing it from being applied to real-life programs. This survey of related work shows that no WCET minimizing unified ILP-based SPM allocation scheme for program code currently exists, whereas basic techniques for program data already exist. For this reason, we focus on SPM allocation of code.

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 SPM allocation presented in this paper.

3. ILP FOR PROGRAM CODE SCRATCHPAD ALLOCATION

This section presents our optimal WCET-aware SPM allocation for program code. Section 3.1 discusses those parts of the ILP modeling a function's control flow. Section 3.2 deals with SPM allocation of consecutive sequences of basic blocks. Modeling a program's global control flow, capacity constraints and objective function are subject of Sections 3.3, 3.4 and 3.5, respectively.

3.1 Modeling of a Function's Control Flow

In the following, ILP variables are represented using lowercase letters whereas constants use uppercase letters. The ILP allocating program code to the SPM uses one binary decision variable x_i per basic block b_i of a program. x_i specifies whether block b_i is allocated to the main memory (mem_{main}) or to the SPM (mem_{spm}):

$$x_i = \begin{cases} 1 & \text{if basic block } b_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if basic block } b_i \text{ is assigned to } mem_{main} \end{cases} \quad (1)$$

Each basic block b_i of a function F causes some costs c_i. These costs reflect the WCET of b_i depending on whether b_i is executed from main memory or from the SPM:

$$c_i = C^i_{main} * (1 - x_i) + C^i_{spm} * x_i \quad (2)$$

For reducible CFGs, an innermost loop L of F has exactly

one back-edge turning it into a cyclic graph. Not considering this back-edge turns L's CFG into an acyclic graph. This acyclic graph without L's back-edge is denoted as $G_L = (V, E)$ in the following. Without loss of generality, it can be assumed that there is exactly one basic block b^L_{exit} in G_L being the loop's unique exit node and one unique entry node b^L_{entry}. The WCET w^L_{exit} of b^L_{exit} is equal to the costs of b^L_{exit}:

$$w^L_{exit} = c^L_{exit} \tag{3}$$

The WCET of a path leading from a node b_i of G_L different from b^L_{exit} to b^L_{exit} must be greater than or equal to the WCET of any successor of b_i in G_L, plus the costs b_i causes:

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i \tag{4}$$

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:

$$c_L = w^L_{entry} * C^L_{max} \tag{5}$$

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 (3) and (4) 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_i has more than one successor because only there, forks in the control flow are possible. Since constraint (4) is formulated for each successor of block b_i, variable w_i always reflects the WCET of any path starting from b_i – irrespective of the fact which of the successors actually lies on the current WCEP. This way, constraint (4) realizes the implicit consideration of WCEPs and their changes in the ILP.

The structure of the ILP constraints of Equations (2) – (5) was originally proposed by [12]. However, these basic constraints of Suhendra et al. need to be refined substantially in order to obtain a functional scratchpad allocation technique for program code. Our extensions to the original ILP formulation are described in the following sections.

3.2 Allocation of Consecutive Basic Blocks

The binary decision variables x_i allow to place a basic block b_i in the SPM independent of allocation decisions concerning any other basic block within a function F. However, this independence of the allocation decisions for single basic blocks is particularly problematic for embedded processors.

If a basic block b_i is allocated to the main memory and an immediate successor b_j of b_i within the CFG is allocated to the SPM, jump instructions must ensure that b_j is reached from b_i. Due to the limited displacement which can be encoded as branch target of typical jump instructions, and due to the usually too large distance between the address spaces of SPM and main memory, a single jump instruction is often insufficient to transfer control from b_i to b_j. Frequently, the address where to jump needs to be computed and stored in an address register so that a register-indirect jump instruction can finally be issued. In such a scenario, transferring control from b_i to b_j requires several machine instructions constituting a severe jumping overhead.

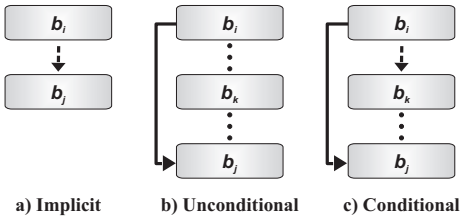

Figure 1: Typical Jump Scenarios

This jumping overhead might be avoided if both b_i and b_j are allocated to the same memory. Thus, the ILP should consider this kind of jumping overhead and should try to allocate groups of consecutive basic blocks to the same memory if this helps in reducing jumping overhead.

On typical embedded processors, three different *jump scenarios (JS)* can be found (cf. Figure 1). If control flows from basic block b_i to b_j without any jump instruction at the end of b_i, this is called *implicit jump*. If a jump instruction at the end of b_i always transfers control from b_i to b_j, this is called *unconditional jump*. Finally, a *conditional jump* transfers control conditionally from b_i to either b_j using an unconditional jump or to b_k via an implicit jump scenario.

The variables x_i, x_j and x_k for the basic blocks b_i, b_j and b_k resp. now provide the information whether jumping overhead needs to be considered within the ILP or not.

If two blocks b_i and b_j belong to the implicit jump scenario and if b_i and b_j are placed in different memories, a penalty should be added since this situation leads to a large jumping overhead to transfer control from b_i to b_j across the different memories. In contrast, no penalty needs to be considered at all if both b_i and b_j are placed in the same memory, because it is ensured that b_i and b_j are allocated adjacently so that no jump is required. We thus define a jump penalty for implicit jumps between basic blocks b_i and b_j as follows:

$$jp^i_{impl} = (x_i \otimes x_j) * P_{high} \tag{6}$$

In Equation (6), the operator $\otimes$ represents the Boolean XOR of two binary decision variables – the Boolean XOR can be modeled within an ILP, but we omitted the listing of these constraints for the sake of simplicity. P_{high} is a constant realizing a high penalty for jumps across the different memories due to their large jumping overhead.

An unconditional jump from b_i to b_j usually bypasses a number of other basic blocks b_k (cf. Figure 1b). These other blocks b_k also need to be considered, because they determine if a jump from b_i to b_j is required at all. If b_i and b_j are allocated to different memories, the high penalty P_{high} already used in Equation (6) needs to be used. If b_i and b_j are allocated to the same memory *mem*, and if no other basic block b_k originally lying between b_i and b_j is allocated to *mem*, b_i and b_j are adjacent within *mem*. Hence, no jump from b_i to b_j is required at all and thus no penalty within the ILP is necessary. If any other basic block b_k is placed between b_i and b_j in *mem*, an unconditional jump from b_i to b_j bypassing b_k is necessary which is penalized by a constant P_{low} which is much lower than P_{high}. The jump penalty for unconditional jumps between basic blocks b_i and b_j is thus defined as follows:

$$jp^i_{uncond} = (x_i \otimes x_j) * P_{high} + \overline{(x_i \otimes x_j)} * (1 - \prod_{b_k \in \text{Figure 1b)}} (x_i \otimes x_k)) * P_{low} \tag{7}$$

In analogy to Equation (6), we omit the description of the ILP equations to model the product of XOR terms.

Since a conditional jump can be seen as the combination of an implicit and an unconditional jump (cf. Figure 1c), the jump penalty for conditional jumps is the combination of Equations (6) and (7):

$$jp^i_{cond} = (x_i \otimes x_k) * P_{high} + (x_i \otimes x_j) * P_{high} + \overline{(x_i \otimes x_j)} * (1 - \prod_{b_k \in \text{Figure 1c)}} (x_i \otimes x_k)) * P_{low} \quad (8)$$

Depending on the jump scenario of a basic block b_i, the overall jump penalty jp_i is defined as follows:

$$jp_i = \begin{cases} jp^i_{impl} & \text{if JS of } b_i \text{ is } \textit{implicit} \\ jp^i_{uncond} & \text{if JS of } b_i \text{ is } \textit{unconditional} \\ jp^i_{cond} & \text{if JS of } b_i \text{ is } \textit{conditional} \\ 0 & \text{else} \end{cases} \quad (9)$$

This jump penalty is used to extend the basic control flow constraints defined in Equations (3) and (4):

$$w^L_{exit} = c^L_{exit} + jp^L_{exit} \quad (10)$$

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i + jp_i \quad (11)$$

3.3 Modeling of the Global Control Flow

Up to this point, the ILP defined in Equations (1) – (11) only models the intra-procedural control flow of a single function F. Without loss of generality, each function F has one dedicated entry block b^F_{entry}. For b^F_{entry}, the ILP variable w^F_{entry} denotes the WCET of any path starting at b^F_{entry} under the assumption that F is called exactly once.

However, some basic block b of a function F' may contain a call of a function F. In this situation, F's WCET represented by variable w^F_{entry} needs to be added to the WCET of block b. In addition, a function call penalty needs to be added to b's WCET since branching overhead similar to that one described in Section 3.2 occurs if b and b^F_{entry} are allocated to different memories. As a result, the overall function call penalty cp_i for a basic block b_i is defined as follows:

$$cp_i = \begin{cases} w^F_{entry} + (x_i \otimes x^F_{entry}) * P_{high} \\ \quad +(1 - (x_i \otimes x^F_{entry})) * P_{low} & \text{if } b_i \text{ calls } F \\ 0 & \text{else} \end{cases} \quad (12)$$

In analogy to Section 3.2, cp_i is used to extend the control flow constraint defined in Equation (11):

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i + jp_i + cp_i \quad (13)$$

Equation (13) now reflects the constraint which is finally generated for our ILP per basic block b_i and per successor b_{succ} of b_i. In Equation (13), we assume non-recursive functions. Due to the practical irrelevance of recursion for embedded real-time software [4], this assumption is valid even if support of recursion could be added to the ILP.

3.4 Scratchpad Capacity Constraints

To obtain a valid SPM allocation, it must be ensured that the size of all basic blocks allocated to the SPM does not exceed the SPM's capacity. Previous work on ILP-based SPM allocation either assumed constant basic block sizes or performed an exponential enumeration of the power set of all basic blocks to model basic block sizes.

As already discussed in Section 3.2, different kinds of jump instructions need to be issued, depending on the contents of the decision variables x_i and x_j for a basic block b_i transferring control to b_j. No jump instruction is needed in situations where b_i and b_j are adjacently placed in the same memory. A conventional jump instruction needs to be generated if b_i and b_j are allocated to the same memory but not adjacently. Finally, complex address computations and register-indirect branches need to be generated if a jump across memories needs to be performed.

Obviously, these different situations have an impact on the size of a basic block b_i. Hence, a block's size depends on the ILP decision variables in practice. In order to cope with such variable block sizes, we fall back to the jump scenarios introduced in Section 3.2 (cf. Figure 1). In the ILP, we only consider b_i's size if b_i is placed in the SPM since we assume a main memory which is large enough to hold the entire program. For a block b_i placed in the SPM, a new variable s_i denotes the growth in size of b_i in bytes if the successors b_j of b_i are kept in main memory. Depending on the jump scenario of b_i (implicit, unconditional or conditional), or if b_i contains a function call, s_i is computed as follows:

$$s_i = \begin{cases} (x_i \wedge \overline{x_j}) * S_{impl} & \text{if JS of } b_i \text{ is } \textit{implicit} \\ (x_i \wedge \overline{x_j}) * S_{uncond} & \text{if JS of } b_i \text{ is } \textit{uncond.} \\ (x_i \wedge \overline{x_k}) * S_{impl} + & \text{if JS of } b_i \text{ is } \textit{cond.} \\ \quad (x_i \wedge \overline{x_j}) * S_{uncond} & \\ (x_i \wedge \overline{x^F_{entry}}) * S_{call} & \text{if } b_i \text{ calls } F \\ 0 & \text{else} \end{cases} \quad (14)$$

For each jump scenario, dedicated constants S_{impl}, S_{uncond} and S_{call} are used. They represent the growth of b_i in bytes for the different jump scenarios. Using s_i, the scratchpad capacity constraint ensuring the validity of an SPM allocation is defined as follows:

$$\sum_{b_i} (S_i * x_i + s_i) \leq S_{spm} \quad (15)$$

In Equation (15), the constant S_i denotes the byte size of b_i in its original form without any cross-memory jumps. S_{spm} represents the available SPM size in bytes.

3.5 Objective Function

The overall goal of our ILP is to minimize a program's WCET by assigning basic blocks to the SPM. Due to the nature of Equations (12) and (13), variable w^F_{entry} corresponds to the WCET of function F including the WCETs of all functions called by F, plus some abstract jump penalties. Since function `main` is the unique entry point of an entire program, variable $w^{\texttt{main}}_{entry}$ denotes the WCET of a program including all penalties. As a consequence, the value of this decision variable needs to be minimized by the ILP:

$$w^{\texttt{main}}_{entry} \rightsquigarrow min. \quad (16)$$

4. COMPILER INFRASTRUCTURE

To turn the ILP model presented in Section 3 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* [5] for the Infineon TriCore TC1796 processor (cf. Figure 2) to extract all the constants required by the ILP from the code currently under optimization.

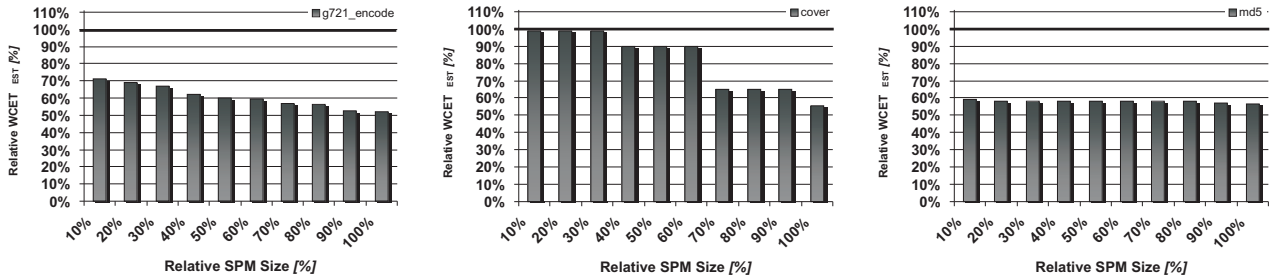

Figure 3: Relative WCET Estimates after WCET-aware SPM Allocation for Representative Benchmarks

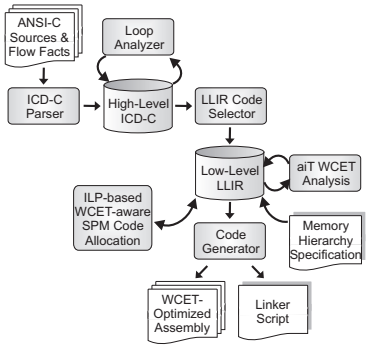

Figure 2: WCET-aware C Compiler WCC

Equation (2) depends on the constants C^i_{main} and C^i_{spm} representing the WCET of basic block b_i if it is located either in main memory or in the SPM, respectively. C^i_{main} is obtained by placing the whole program in main memory and performing a WCET analysis using the coupling of our compiler backend to the static WCET analyzer aiT [1]. In a second step, the whole program is virtually placed in the SPM using an infrastructure to model arbitrary memory hierarchies within our compiler. Another WCET analysis of this program yields the values C^i_{spm}.

Equation (5) depends on a loop's maximal iteration count C^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 [8]. Irrespective of the origin of C^L_{max}, flow fact mechanisms take care to keep the values C^L_{max} up to date during all loop optimizations of our compiler such that correct values are used by our ILP for SPM allocation.

The jump penalties P_{high} and P_{low} introduced in Section 3.2 do not rely on any part of our compiler infrastructure. WCET analyses of jumping code for the different jump scenarios revealed that the values 16 and 8 are appropriate for the considered TriCore architecture.

Equation (14) depends on constants S_{impl}, S_{uncond} and S_{call} representing the byte size of the additional code required to jump from a block b_i to b_j if b_i is allocated to the scratchpad memory but b_j is not. Due to the shape of the jumping code required for the TriCore architecture and the different jump scenarios of Equation (14), S_{impl} and S_{uncond} equal to 10 bytes and S_{call} is equal to 12 bytes.

A basic block's size S_i (cf. Equation (15)) without consideration of cross-memory jumps at the end of b_i is simply computed by accumulating the size of all instructions of b_i. The totally available SPM size S_{spm}, however, is extracted again from WCC's memory hierarchy infrastructure.

After solving the ILP, the values for the decision variables x_i determine where to place each basic block b_i. Using WCC's memory hierarchy API, the code of the program currently under optimization is finally transformed such that it reflects exactly the allocation decisions taken by the ILP. In addition to the SPM-allocated assembly code, our compiler finally emits a linker script required to generate a binary executable reflecting the ILP's SPM allocation.

5. EVALUATION

This section presents real-life benchmarking results for the proposed optimal WCET-aware SPM allocator. At optimization level *-O2*, our compiler (cf. Figure 2) applies a total of 34 different optimizations, including, among others, code reordering transformations. As very last optimization, the WCET-aware SPM allocation of program code discussed in this paper is performed. Hence, our SPM allocation is always applied to already highly optimized code.

As target architecture, the Infineon TriCore TC1796 is considered. Its memory address space is divided into 16 segments of maximum size of 256 MB each. The instruction set allows to jump within a segment with only one single machine instruction. Cross-segment jumps require more instructions (cf. Section 3.2). The TC1796 features a 48 kB large program code scratchpad mapped to segment 13. From these 48 kB, 1 kB is reserved for system code so that 47 kB remain for free use. An access to the program SPM of the TC1796 takes place within one cycle whereas accessing the program Flash serving as main memory takes 6 cycles.

Our ILP-based SPM allocation was applied to a total of 73 different real-life benchmarks from the MRTC [9], MediaBench [7], UTDSP [13] and DSPstone [15] benchmark suites. The number of basic blocks of all considered benchmarks ranges from 4 for the simplest DSPstone codes up to 585 for the most complex MediaBench applications. The code sizes of the benchmarks range from 52 bytes up to 18 kB with an average code size of 2.8 kB per benchmark.

Since the benchmarks' code sizes are considerably smaller than the totally available SPM size, we artificially limit the available SPM size for benchmarking. For every single benchmark, SPM sizes of 10%, 20%, ..., 100% of the benchmark's code size were used. The following results show the WCET estimates of all benchmarks produced by aiT resulting from our WCET-aware SPM allocator as a percentage of the WCET when not using the SPM at all.

Figure 3 shows the impact of our WCET-aware SPM allocation on the WCET estimates *($WCET_{est}$)* of three representative benchmarks. For `g721_encode` with a total code size of 3,204 bytes, a steady decrease in terms of WCET can be observed the more scratchpad is available. Already for extremely small SPMs of only 10% of the program's code size, the WCET after our optimization amounts to 71% of the original WCET, i. e. a WCET reduction of 29% was achieved. If the benchmark fits into the SPM in its entirety,

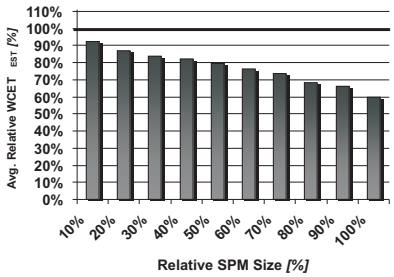

Figure 4: Average WCET Estimates after WCET-aware SPM Allocation for 73 Benchmarks

the resulting WCET is only 52.2% of the original WCET leading to savings of 47.8%.

For `cover` (code size: 2,670 bytes), a stepwise WCET reduction was observed. At 40%, 70% and 100% of SPM size, our ILP is able to move the most important loops leading to the highest WCET savings entirely onto the scratchpad memory. Thus, WCETs of 89.8%, 65.1% and 55.7% were achieved for these SPM sizes, resulting in savings of 10.2%, 34.9% and 44.3% respectively.

A quite extreme evolution of WCETs was finally observed for `md5` (6,354 bytes size). Here, an SPM size of only 10% of the total size of `md5` is enough to reduce the benchmark's WCET down to 59.1%. In absolute values, a tiny SPM of 636 bytes leads to a WCET reduction of 40.9%. This indicates that `md5` consists of one single hot spot which is executed extremely frequently and which has a very small code size. A further increase of the SPM for `md5` does not translate into any further WCET reductions. If `md5` fits entirely into the SPM (100%), an overall WCET of 56.3% was obtained which is only 2.8% off the WCET obtained for the 10% SPM. This benchmark clearly demonstrates that our ILP unerringly selects those basic blocks leading to the highest WCET reductions when being moved onto the SPM.

On average over all 73 benchmarks, we finally obtained steadily decreasing WCETs with increasing SPM sizes (cf. Figure 4). Already for small SPMs, WCETs decrease to 92.6% of the WCET without any SPM, corresponding to a WCET reduction of 7.4%. For large SPMs storing the entire benchmark, average WCETs of only 60% of the original WCET were obtained, leading to overall savings of 40%.

The complexity of our proposed ILP-based SPM allocator is of no practical relevance. For a CFG with n nodes, the ILP defined in Section 3 has a size of $\mathcal{O}(n^2)$ constraints and variables. However, the ILP solver *cplex* only takes one or two CPU seconds on an Intel Xeon running at 2.4 GHz for each of the 73 benchmarks. Compared to this, the two WCET analyses required to generate the constants C^i_{spm} and C^i_{main} (cf. Section 4) are more expensive, but they also terminate within a few CPU minutes for our largest benchmarks.

6. CONCLUSIONS

This paper is the first one to present a technique for optimal WCET-aware SPM allocation of program code. It improves the current state of the art of SPM allocation in that it performs ILP-based SPM allocation of code for the very first time under consideration of jump penalties and variable basic blocks sizes based on jump scenarios. The effectiveness of our approach is demonstrated by WCET reductions from 7.4% up to 40% for 73 different real-life benchmarks.

Our future work will concentrate on developing ILP-based scratchpad memory allocators for program data and on dynamic SPM allocation of both code and data where changing the scratchpad's content at run time will be possible.

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] A. M. Campoy, I. Puaut, A. P. Ivars, et al. Cache contents selection for statically-locked instruction caches: An Algorithm Comparison. In *Proceedings of ECRTS*, Palma de Mallorca, July 2005.

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

[4] J. Engblom. Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis. In *Proceedings of RTAS*, Vancouver, 1999.

[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] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO 30*, Washington DC, 1997.

[8] 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*, Mar. 2009.

[9] Mälardalen WCET Research Group. WCET Benchmarks. *www.mrtc.mdh.se/projects/wcet*, Sept. 2008.

[10] I. Puaut. WCET-centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proceedings of ECRTS*, July 2006.

[11] S. Steinke, L. Wehmeyer, B.-S. Lee, et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of DATE*, Paris, Mar. 2002.

[12] V. Suhendra, T. Mitra, A. Roychoudhury, et al. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of RTSS*, Miami, Dec. 2005.

[13] UTDSP Benchmark Suite. *www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html*, Sept. 2008.

[14] M. Verma and P. Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer, 2007.

[15] V. Živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of ICSPAT '94*, Dallas, 1994.

[16] L. Wehmeyer and P. Marwedel. Influence of Onchip Scratchpad Memories on WCET Prediction. In *Proceedings of WCET*, Catania, June 2004.

[17] L. Wehmeyer and P. Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In *Proceedings of DATE*, Munich, Mar. 2005.

[18] L. Wehmeyer and P. Marwedel. *Fast, Efficient and Predictable Memory Accesses – Optimization Algorithms for Memory Architecture Aware Compilation*. Springer, 2006.