

TUM

INSTITUT FÜR INFORMATIK

Improvements on the cache behaviour prediction by modular arithmetic

Flexeder Andrea, Petter Michael, Seidl Helmut



TUM-I0933
November 09

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-11-I0933 -0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2009

Druck: Institut für Informatik der
 Technischen Universität München

Improvements on the cache behaviour prediction by modular arithmetic

Andrea Flexeder, Helmut Seidl and Michael Petter

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany,
{flexeder, seidl, petter}@cs.tum.edu,
WWW home page: <http://www2.cs.tum.edu/~{flexeder,seidl,petter}>

Abstract. In this paper we present the results of our *improved value analysis* of modular integer arithmetic in order to obtain more precise information about memory accesses and the alignment of cache lines. This analysis was developed, implemented and evaluated within the *SuReal project* [1]. The aim was improving WCET computation by extending the value analysis in the WCET analyser *aiT*[5] by modulus information and linear relations [9] between processor registers and memory locations. This relational information allows for at least alignment information and hints about the cache hit rate if no concrete value information about processor registers and memory locations is available. The focus of this paper is on drawing a comparison between the value analysis without modulus information as is currently implemented in *aiT* and our improved value analysis by means of our benchmark suite. Concluding, the results inferred by the improved value analysis allow for a more precise and reliable check of alignment properties in the data cache, rendering WCET computation more precise.

1 Introduction

Especially in the field of safety critical real-time applications there arises the need to verify that all timing constraints on the considered system will be adhered. The fact that the program will meet its deadlines is as important as the correctness of the program. Deriving run-time guarantees (i.e. that each task finishes before its deadline) for real-time systems has thus attracted more attention in the area of static program analysis. In this regard it is intended to statically determine the worst-case execution time (WCET) of a program. In order to achieve reliable assumptions about this maximal time taking for a certain program to execute on a specific system, one has to analyse the executable instead of the source code. The program analyser must only rely on the executable and cannot be sure that the execution of the source code by the processor really behaves as the code writer intends to, according to the principle WYSINWYX [3].

The technique of abstract interpretation allows deriving statically safe WCET bounds on the timing model of the processor. With this timing analysis it is possible to verify the compliance with timing constraints of the tasks within safety critical real-time systems. Recently, the gap between processor speed and memory access is growing. Thus, cache memories are applied to provide faster access to recently referenced memory regions. Despite this performance benefit, caches in real-time systems show up the drawback in complicating a reliable and accurate WCET prediction, because the performance of

the cache appears quite unpredictable. Since the performance of the cache has to be considered, many program analyses aim at providing a reliable and hopefully precise approach to predict cache hit and miss rates for real-time systems in order to render the computation of WCET bounds more precisely.

In the following we shortly describe the WCET analyser from AbsInt and the concept and an evaluation of our improved value analysis. By means of our benchmark suite we present the results of our improved value analysis, enriched with modulus information, compared to the value analysis from *aiT* without modulus information.

aiT framework. *aiT*, the timing analyser conducted by AbsInt, provides safe upper bounds for the WCET of an executable. The WCET analysis in *aiT* is separated into the following phases [5]:

- *value analysis*: computing value ranges for processor registers and memory locations
- *cache analysis*: ranking memory references as cache miss or hit
- *pipeline analysis*: predicting program behaviour on the processor pipeline
- *path analysis*: determining the WCET path of the program

This analysis framework is illustrated in figure 1. Here, we shortly mention some characteristics of the underlying *aiT* framework. The input to be analysed is a fully linked executable program without any information about program and data allocation. Thus, the executable is fed into a parser, which reconstructs the basic control flow structure, yielding one *interprocedural control flow graph* [2]. In order to be able to compute a safe upper bound for the WCET of a program, it is necessary to consider both the program to be analysed and the underlying hardware. Thus, the instruction set of the processor and the code generation scheme of the used compiler have to be taken into account, in order to annotate each control flow edge by the corresponding processor instruction. In the following and for our analysis, we restrict the underlying hardware to the 32-bit *Power PC* architecture (*PPC*)[7], which is primarily applied in control applications in the area of embedded safety critical systems [8]. In the *aiT* analysis framework for WCET computation the *call string approach* is used for constructing an interprocedural analysis. In this regard the whole history of a function call is embedded into an abstract program state. This approach allows a separate analysis of different call sequences, since these different call contexts can be distinguished statically.

Note that the *value analysis* is the source of providing information about the values stored in processor registers and memory locations for every program point and execution context. The existing approach of the value analysis in *aiT* is based on a simple interval analysis, providing at every program point an interval of possible values occurring during runtime (for details see [5]) for every processor register and some memory locations (primarily for those which are specified by effective addresses). These results provide the basis for identifying possible addresses of indirect memory accesses within the cache analysis.

Now, in order to improve WCET computation even if no concrete values about processor registers and memory is known, it is our aim to improve the existing value analysis by inferring *linear modulus relations* between processor registers and memory locations.

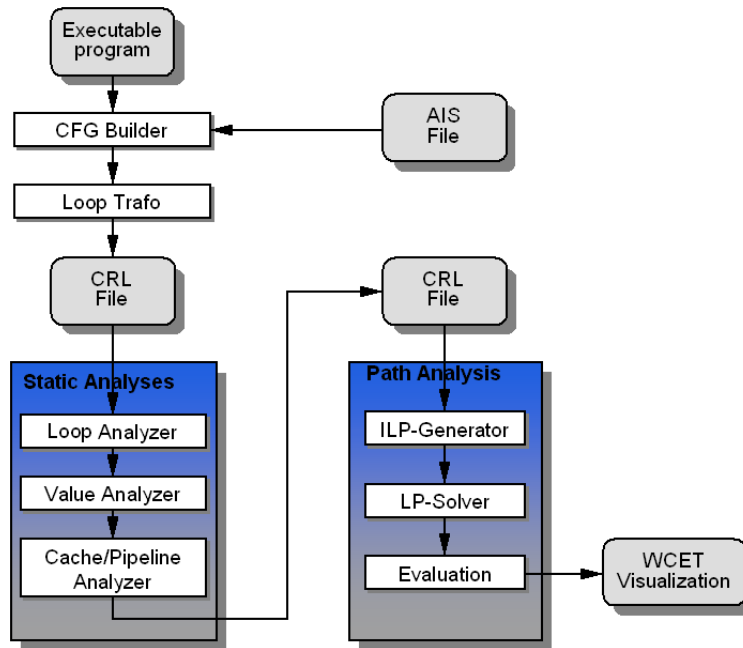


Fig. 1. WCET computation phases

The following section 2 addresses a short overview of our *improved value analysis* which is partitioned into the following phases:

1. the classification of memory locations within the executable,
2. the inference of linear relations between registers and memory locations
3. and an interval analysis to provide registers and classified memory locations with possible contents along all states and execution traces along the program.

2 Improved Value Analysis

Our improved value analysis was developed within the *SuReal project* [1] with the objective of specifying a preferably precise analysis of memory accesses in executables. Since we implemented our approach in the *aiT* framework, we draw upon the *call string approach of length zero*. The overall goal is to infer information about data alignments, classifying memory locations and the assignment of memory accesses to cache lines. In this context we are especially interested in inferring linear equality relations, since considering loop iterations the accessed memory cells within the loop body are linearly dependent of the loop iteration variable, which mostly is a local variable. Additionally these linear relations contribute in identifying indirect accesses to local variables in the cache. By applying modular arithmetic, modulus information can be inferred for stating alignment properties. Thereby, we expect to be able to directly classify more memory accesses as stack or heap related and at least stating alignment information if no concrete values of processor registers and memory locations are known. Here we present a short overview of the single phases of our improved value analysis.

Classifying memory locations. Tasks spend most of their execution time in loops and recursive functions and therefore it is an essential task to determine bounds on the iterations of loops and on the recursion depth of functions. A necessary ingredient for this are the values of variables, registers or memory cells occurring in conditions tested for the termination of loops or recursion. Consequently the method for data-cache behaviour analysis needs to know effective memory addresses of data to determine where a memory access goes to. Thus, the first analysis phase addresses the problem of identifying candidates for *local variables*. Additionally local variables have a considerable impact on statements about indirect memory accesses. Hence, a classification of memory locations results in more precise analysis results of the subsequent cache analysis. Note that the address of a local variable is linearly dependent of the initial value of the stack pointer register. Consequently, for the identification of candidates for local variables, we preface an analysis inferring all valid linear equality relationships between the processor registers, which tags all stack pointer relative accesses as possible local variables. Since processors perform their arithmetic modulo a power of two, additionally we take modulus information into account. Thus, this analysis is designed over the ring $\mathbb{Z}_{2^{32}}$ (for the PPC architecture). For more details about our approach classifying memory locations, refer to [6].

Summarising, this analysis infers candidates for local and global variables in form of a normalised representation, i.e. every access to a possible local memory location is described by an offset relative to the beginning of the actual stack frame.

Linear relations between registers and memory locations. As we have identified candidates for local and global variables, we are able to correlate them to the processor registers, i.e. we perform a linear equality analysis, providing information about the relationships of registers and the classified memory locations. Especially in safety critical executables, where an optimisation level of zero or even less was chosen, it is often the case that the values of local variables are saved in memory because of the finite number of processor registers and the coding conventions for function calls. Thus, in order to retain information about memory locations, the relation between processor registers and memory locations has to be taken into account. We take advantage of this kind of information both when inferring alignment properties and in the interval analysis furnishing every classified memory location with an interval of possible values [6].

Interval analysis. In real-time systems many loops contain array operations. When considering the iterations of loops, the array cells within the loop are often accessed relative to the loop counter variable, which mostly turns out to be a local variable. In order to recognise accesses to the single array elements, the bounds of the loop counter are required. When the bound of the loop counter variable is known, we can infer a bound for the accessed memory cells. Thus, an interval analysis is required which computes for each processor register and the classified memory locations an approximation of all possible values (i.e. concrete values or addresses) occurring during runtime.

Alignment properties. As stated in [11] the alignment of memory accesses has a crucial impact on the performance of memory accesses and thus on the computation of the WCET. For this purpose we are interested in as precise alignment properties as possible. For WCET computation we concentrate on the aspect of *memory operand alignment*. As described in [7], according to the length of the operand, every operand of a single-register memory access instruction has a natural alignment boundary. This means the address of an aligned operand is an integral multiple of its length. When accessing memory operands the best performance can only be guaranteed if alignment on their natural boundaries is given – otherwise one may meet performance degradation. This alignment property means: an access to an operand of size n bytes at byte address a is aligned iff $a \bmod n \equiv 0$. For our analysis this means that when considering memory access instructions the memory access expressions are checked for operand alignment, i.e. divisibility by the length of the operand.

In addition, at every memory access we use the inferred information to check the valid modulo 2^{32} relations (in case of the PPC architecture) for operand alignment w.r.t. the size of the cache line (2^5 for PPC). This allows us determining e.g. within a loop that every n -th memory access is misaligned. With this knowledge deduced so far the additional cache analysis[5] is refined. In the cache analysis the accesses to main memory are classified as cache miss, cache hit or unclassified access. The alignment properties, the inferred linear equality relations and the values of memory locations contribute in categorising a memory access as miss or hit – this cache hit information is used in the *pipeline analysis* to eliminate pipeline stalls [5].

Summarising, all this accumulated information about the registers and memory locations – their contents and their linear equality relations with modulus information – is used for improving the identification of memory accesses, i.e. to resolve unknown memory accesses in the cache analysis. One consequence is that within the pipeline analysis accesses to the same cache lines can be merged and thus improve on WCET computation.

In the next section we present experimental results and an evaluation of the implementation of our improved value analysis.

3 Experimental Results

Within this section first we present our results by means of some simple test programs. Then, we contrast the value analysis *without* modulus information with those *with modulus information* by means of some real-world example programs (this means program fragments from avionics and automobile control software).

Proprietary test programs

We conducted a test series on a 2,2 GHz Opteron machine equipped with a physical memory of 16GB. Our analysis providing affine equality relation and interval information for processor registers and memory locations, quickly terminates in the range of a few seconds. Only program *edn* takes much more time and memory compared to the other example programs. However, this program makes extensive use of local arrays, reference parameters and pointer arithmetic. All the example programs are compiled with gcc with optimisation level zero for the PPC architecture. The following table shows some practical results of our prototypical implementation.

Program	Procs	Instr	MemAcc	Unknown	Locals	Globals	PosUnal	Time	MemUse
prime	11	250	48	34	14	–	1	5.27sec	358MB
edn	16	1153	442	68	374	–	62	2024.53sec	3276MB
standard	7	116	48	–	17	–	–	0.91sec	93 MB
switch	1	88	25	–	25	–	–	1.66sec	74MB
mod	1	33	16	2	14	–	2	0.97sec	70MB
brake	6	212	112	12	44	17	–	4.54sec	382MB
top	14	1822	827	4	17	–	–	0.96sec	0,06MB

Table 1. Some results of our test suite

Within this table we specify:

1. the number of functions *Procs* and instructions *Instr*
2. the number of memory access instructions *MemAcc*,
3. the number of memory accesses our analysis was not able to classify *Unknown*,
4. the number of possible candidates for local *Locals* and global variables *Globals*,
5. the number of memory accesses which may be unaligned *PosUnal*,
6. the time *Time* and memory requirements *MemUse* of our analysis run.

Our test programs are available online <http://www2.in.tum.de/flexeder/tests>.

Here, we present an extract from our test suite. The first two programs *prime* and *edn* are example programs from the WCET suite within the framework PAG. The program *standard* is generated from C code which conforms to the following characteristics:

- no dynamic memory allocation
- no pointers/pointer arithmetics
- no recursion or unbounded loops
- no local arrays

This is no special case. Especially in the area of software development for avionics restrictive safety guidelines have to be kept, as e.g. the SW standard DO-178B [10]. Many automobile and aviation software development companies demand that the produced code is simple, deterministic and efficient and should require as few resources as

possible in terms of memory and execution time [4]. The objective of this guidelines is ensuring that SW performs its intended function with a level of confidence in safety. In order to ease the certification process the considered C code, which is mostly generated automatically from high-level specification languages as.g. SCADE or SIMULINK in the area of safety-critical code development, should show the characteristics listed above. This means within the program *standard* only the semantical program components of local and global variables, global arrays and control structures occur. Programs *mod* and *switch* are in order to test alignment properties and finally the two programs *brake* and *top* generated via Scade for our SuReal demonstrator [1].

When we consider only assembly code conforming to these characteristics all the memory accesses to local and global variables were precisely identified, as e.g. the table entry for program *standard* specifies. This involves that the alignment property of memory accesses can be precisely classified as *aligned* or *not aligned*. In case of programs *mod* and *switch*, we also take the concept of local arrays and pointers into account. In case of *mod* there are two memory accesses marked as *possibly unaligned*.

```

...
int *p, i;
for (i=0; i<10; i++, p++)
    *p = i;
...

```

We obtain this imprecise result since there is a *store*-instruction to an unknown memory location. There, we discard all information about the local variables so that even the alignment information gets lost.

In presence of the concept of pointers and pointer arithmetic, our approach is not able to precisely handle reference parameters. There all information about the local variables gets lost. As in case of an unknown target address of a *store*-instruction and passing a pointer as parameter to a function call, we destroy all information about the local variables to provide a safe analysis.

Real-world test programs

In the following we present some results where our improved value analysis was applied to some real-world example programs from automobile and airbus supplier companies. The tests were conducted by AbsInt – the testing environment was the same as described in [12] – a 2,6 GHz Core2Duo machine with 8GB RAM. Only the binary was provided which was compiled with a very conservative optimisation level (gcc-level O0 or even less). Note, that the example programs are quite small, since complex programs were split in order to obtain WCET results for single program fragments. The runtime of the single WCET analyses was in the range of a few seconds until maximally one hour. The following table draws a comparison between the old value analysis by AbsInt [5] without modulus information ($WCET_{old}$) and our prototypical implementation [6] ($WCET_{new}$) by means of the component WCET (measured in clock cycles).

Program	$WCET_{old}$	$WCET_{new}$
1. Avionics Supplier 1		
Init STM	290	251
Alpha + Beta	1980	1846
Init GRP	1274	1143
Kinematic UPD	6734	6414
Fuel ESTM	8757	8331
Navigation UPD	9680	9662
Omega	26845	25485
U-Switching	33880	33225
V-Monitor	88766	85324
Navigation Cons	312083	300566
v116	144414	99125
v152	435039	428510
Calculation PA	69325	66171
2. White+Yellow Products Manufacturer		
Modus 0	2971972	2950245
Modus 2	742600	738825
Modus 3	308528	306583
Modus 4	1288638	1283103
Modus 8	1964928	1957582
3. Engine Manufacturer		
Task 1300	1820885	1640018
Task 1312	2000988	1755024
4. Avionics Supplier 2		
Safety T1	5019785	4096592
Safety T2	5090933	4154095
Safety T3	13390593	10436261
Safety T4	17342887	13818679
Safety T5	13326107	10376288
Safety T6	15193027	11812494
Safety T7	5015843	4093736
Safety T8	6819351	5483274
Safety T9	7747796051	5491645936
5. Automotive		
Task 5ms	201456	156000
6. Avionics Supplier 3		
5D	254910	130267

Fig. 2. value analyses comparison by means of WCET

The provided programs of *Avionics Supplier 1* are built up of many simple sub tasks, which are excessively called by the main program. The modulus information of our improved value analysis allows that some memory accesses can be identified exactly instead of getting an interval of possible accesses. Thus, within the pipeline analysis more accesses to the same cache line can be combined. Additionally these test programs are characterised by containing many loops.

For the second test suite of *White+Yellow Products Manufacturer*, which contains an endless loop, in which via global variables and switch tables in every pass different sub tasks are initiated, we conclude: The modulus information helps in identifying cache misses, since our improved value analysis is able to exactly classify pointers to big data structures. Thus, within the cache analysis we improve on specifying the accesses within a cache line as cache miss or hit.

The following two test cases show Intel *i386* code in the setting of control software for avionics.

Program	$WCET_{old}$	$WCET_{new}$
7. 2000_com		
T1	149499	146835
T2	167020	163392
T3	162275	159304
T4	160402	156648
T5	152935	150148
T6	161960	158380
T7	165990	162483
T8	160767	156078
T9	151697	148229
T10	164247	160363
T11	165008	161566
T12	160936	156464

Program	$WCET_{old}$	$WCET_{new}$
8. 2001_mon		
T1	148496	148029
T2	146709	146129
T3	155778	155354
T4	149753	148752
T5	150716	150150
T6	148961	148393
T7	156558	156159
T8	153410	152415
T9	150885	150466
T10	149718	149153
T11	156475	156076
T12	153858	152753

Fig. 3. value analyses comparison by means of WCET

The tables show that compared to the former value analysis modulus information contribute much in improving WCET computation. Performing *aiT* analysis on these examples is quite fast, yielding analysis results in maximally 3 minutes.

Summarising, it is obvious that the improved value analysis, enriched by additionally providing modulus information, helps in precisely classifying memory accesses. Consequently, within the subsequent cache and pipeline analyses more memory accesses are precisely known, providing information about possible cache misses. Thus, by applying a value analysis with modulus information the WCET information can be partially improved dramatically.

4 Conclusion

In this paper we presented the concept of our *improved value analysis* implemented and integrated in the WCET analysis tool *aiT*. The improvement comprised of extending the value analysis by linear equality relations and modulus information. Especially in the case of loops precision is lost when nothing is known about the values of the loop counter variable. The preceding cache and pipeline analysis do not have any information about memory accesses in the loop. Now, by providing at least relational and modulus constraints more precise statements about cache hits and misses are possible. Additionally it turned out that this proceeding even redundantises user annotations, primarily the address of the stack pointer. As the experimental results show, the prediction of the cache behaviour and the WCET has improved. Furthermore the practical experiments indicate that this approach is quite efficient, providing quite precise results in reasonable time.

However, we want to remark the major limitation of the approach presented here. It is not able to precisely deal with reference parameters, as in this case we take a very coarse approach [6] discarding too much information. Thus, a side-effect analysis to precisely handle reference parameters has to be implemented in a next step.

References

1. Sicherheitsgarantien Unter REALzeitanforderungen. <http://www.sureal-projekt.org/>.
2. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66. Springer-Verlag, 1996.
3. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin, Madison, WI, USA, August 2007.
4. B. Dion. How to meet EUROCAE ED-12B / RTCA DO-178B international software safety regulation for airborne systems while reducing development cost, 2007.
5. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 469–485. Springer-Verlag, 2001.
6. A. Flexeder, M. Petter, and H. Seidl. Analysis of executables for WCET concerns. Technical Report, Institut für Informatik, 2008. Available online <http://www2.in.tum.de/flexeder/report38.pdf>.
7. IBM. *PowerPC Architecture - The official manual for the PowerPC architecture. Three parts: instruction set architecture, virtual environment architecture, and operating environment architecture, IBM book number SR28-5124-00*. 1993.
8. C. King. Power architecture platform - the intersection of technology and brand evolution. 2006.
9. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.
10. RTCA. DO-178B software considerations in airborne systems and equipment certification, 1992.
11. S. Sobek and K. Burke. *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*, 2004. http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf.
12. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2-3):157–179, 2000.