Towards Automatic Translation to Temporally Predictable Code[*]

Robert Staudinger
University of Salzburg
Department of Computer Science
5020 Salzburg, Austria
rstaudinger@cs.uni-salzburg.at

Abstract

Contemporary Microprocessors are highly optimised towards average case performance using caches and branch prediction. While these features provide considerable speedups they come at the price of predictability. However, for real-time applications with timing precision requirements in an order of magnitude close the CPU's clock frequency, tight prediction of WCETs (worst case execution times) is indispensable. We are proposing a conceptual model and an assembly transformation strategy to turn code with nested conditional control structures into code with a flat flow of control. This so-called single-path code facilitates the prediction of timing behaviour, ideally causing only an negligible overall slowdown. To overcome the burden of writing a full fledged compiler, we are designing our transformation to be applied post-pass, with full support for any optimisations conducted during the preceding compilation stage.

1 Introduction

Moore's law does not go past embedded systems. CPUs of all architectures and dimensions are constantly superseded by more powerful successors. However, an unfortunate side-effect anent to the domain of time-critical applications is, that more contemporary microcontrollers tend to expose increasingly non-deterministic behaviour regarding individual instruction latencies. This can largely be attributed to the hierarchical memory model with regard to a program's data, and pipelined execution pertaining to its code. We focus on the latter issue: when branch prediction fails, the pipeline of fetched and decoded instructions has to be flushed and refilled before program execution can proceed. Since branches can – and often will – depend on input data passed to a program at runtime, it is even theoretically impossible to correctly predict them in all and any cases. This poses a problem for hard real-time systems, where tight prediction of a program's timing behaviour is indispensable.

The traditional approach to overcome this problem is to exactly model the hardware in question and apply path analysis to determine worst case timing scenarios [4]. However, a correct simulation of CPU intrinsics, including behaviour like instruction latencies and cache effects, is very specific to the model in question, and thus tied to considerable effort.

In the light of the complexities adherent to prediction of a program's worst case execution time (WCET), Puschner proposed the *Single-Path Approach* [9] to timing-aware algorithms. The essence of this concept is to transform the code from control dependence to data dependence [1], removing conditional branches, and thus eliminating the nondeterminism they are inducting. Single-path algorithms use *predicated instructions* to conditionally execute code instead of branching.

Research on predicated execution has extensively been conducted to increase performance on high-end processors [6]. Their high clock frequencies depend on long pipelines, which in turn increases the performance impact of pipeline stalls. An approximative rule of thumb for conditionally executed sequential blocks of code is, that predicated execution is favourable over branching, if the time required to execute the block is shorter than the time required to recover after a pipeline stall. Predicated instructions propagate through the pipeline just like their unconditional counterparts, but – the depending on CPU architecture – the execute and/or writeback stages are not executed but swapped for NOPs if the assocciated boolean predicate is *false*. Consequently any actual side effects caused by the execution of the instruction in question are impeded.

While the algorithms presented in [9] require manual adoption of source code, we are interested in automatic translation to single-path code. Rather than implementing

[*]This project has been supported by the Austrian Science Fund, project No. P18913-N15

```
void bsort (int a[], int n) {
  int i, j, t;
  for (i = n − 1; i > 0; i − −) {
    for (j = 1; j <= i; j + +) {
      if (a[j − 1] > a[j]) {
        t = a[j];
        a[j] = a[j − 1];
        a[j − 1] = t;
  } } }
}
```

Figure 1. Bubble-Sort Algorithm in C

```
procedure transform (block, predicate) begin
  for each op in block do
    rewrite_predicate (op, predicate);
    if b := get_subordinate_block (op) then
      p := push_predicate (op);
      transform (b, p);
      pop_predicate ();
    end if
  loop
end
```

Figure 2. Transformation Algorithm

a full-blown compiler, we are investigating transformations on assembly level, to allow for building upon already optimised code.

This paper ist structured as follows. Section 2 introduces the *predicate stack* model we conceived for single-path execution of nested control flow graphs (CFGs) and illustrates the transformation using a real-world example. Section 3 presents how we are mapping the model to the ARM instruction set. Section 4 outlines preliminary experimental evaluation of this work in progress, and Section 5 gathers first conclusions and outlines future work.

2 The Predicate Stack Model

In the context of this paper, by referring to conditional blocks of code we are only identifying strictly *forward conditional* ones. We denote a block b_i being forward conditional if it does not have a backwards edge to the immediate predecessor block b_{i-1}. Using this criterion we can sort out conditional blocks induced by loop constructs. For a more thorough discussion treating the reconstruction of CFGs from assembly code we refer to [2].

For automatic translation of arbitrary programs to their semantically equivalent single-path counterparts we introduce the notion of a *predicate stack*. The elements on the predicate stack mirror the nesting of conditional code blocks in the CFG. Conditional branches push onto the predicate stack, the associated join-nodes pop from it. Conditional code executes taking into account the topmost element on the predicate stack.

With regard to the model described in this section either alternatives are equivalent. If a block is already relying on predicated execution (e.g. introduced by an optimising compiler), what is left to do for the translation step is allocating the respective condition register on the predicate stack.

For the purpose of illustrating the transformation strategy and run-time execution mechanism we are using the bubble sort algorithm, also used in [11]. Figure 1 reproduces the source code exactly as used in our experiments. Furthermore Figure 4 (a) shows a terse, simplified CFG, (b) depicts the counterpart CFG after translation to single-path code. The utilization of the predicate stack can be read off at the right of sub-figure (b). Bsort is built around a single conditional block, there is no further nesting. Hence only one predicate is needed to indicate whether the code is actually executed or just passed through the CPU's pipeline without side-effects.

The predicate in question (denoted *p0*) depends on the result of the comparison (Fig. 4, Block 3'). Thus the transformation process has to insert the predicate allocation accordingly. The operations of subordinate Block 4' are predicated with p0. Finally p0 is revoked in Block 5', before the conditional is tested again.

Obviously, in the general case of a nested conditional block b' within a block b, the predicate associated to b' always depends on the the predicate of the surrounding block b, as code within a disregarded branch must never be executed. Therefore each predicate that is pushed on top of a non-empty predicate stack has to be combined with the current top element at program execution time using logical *and* (c.f. Figure 3).

```
procedure push_predicate (op) begin
  new_pred := get_predicate (op);
  if stack_is_empty () then
    stack_push (new_pred);
  else
    cur_pred := stack_top ();
    stack_push (cur_pred ∧ new_pred);
  end if
end
```

Figure 3. Predicate Stack Manipulation

Figure 2 presents the recursive algorithm used to transform a program's CFG (constructed from assembly code) into single-path code. For the sake of brevity and clarity

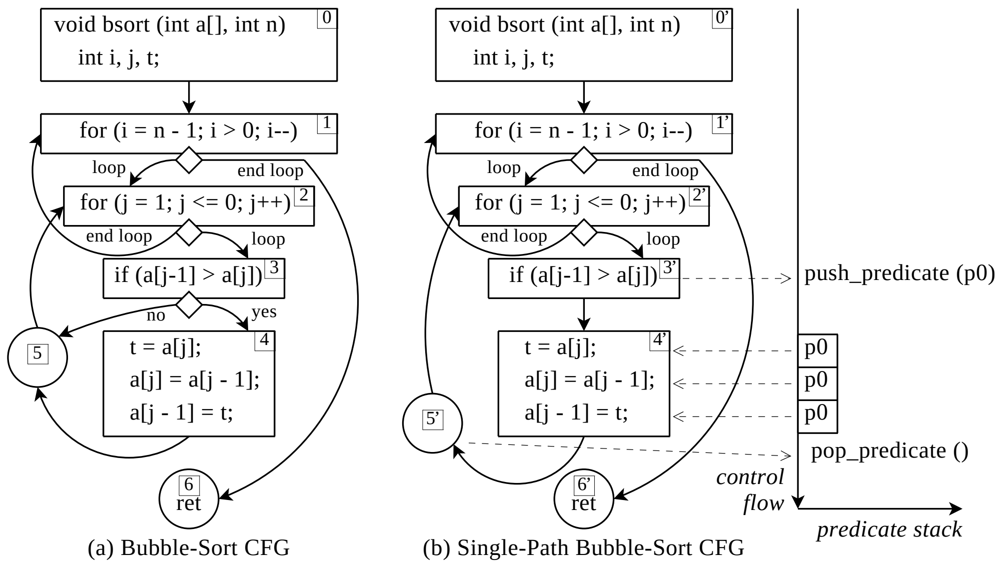

Figure 4. Bubble-Sort CFG Sketch and Single-Path CFG Sketch with Predicate Stack

the special casing for the entry block, which is not associated with a predicate by definition, is omitted. The algorithm transforms each operation in the block to use the assigned predicate (Line 3). In *rewrite_predicate()* two different cases have to be considered. (i) The instruction does not yet have an assigned predicate, in which case it is simply added. (ii) The processed instruction is already predicated as a result of optimisations done by the compiler, the predicate has to be rewritten to use the one currently on top of the predicate stack. In case the CFG forks to subordinate blocks, a new predicate – associated with the currently processed operation – is allocated on top of the predicate stack. The *transform()* procedure recurses to process the new block before the predicate is removed from the stack (Lines 4-7). This results in a depth first traversal of the CFG.

3 Mapping to the ARM Architecture

We are implementing the model proposed in the previous section on an ARM architecture[1] due to the significance this CPU family has for embedded systems appliances. More specifically we are using an XScale PXA255 ARMv5 CPU on a Gumstix Connex board[2].

ARM opcodes fully support predicated execution, therefore the translation of instructions is straightforward. The opcodes in question either have to be rewritten to their predicated counterparts, or in case they are already predicated by virtue of compiler optimisations (e.g. using "-O3" for gcc), the predicate has to be swapped for the respective one topping the predicate stack.

For the representation of the predicate stack at runtime we are using the condition flags provided by the program status register (PSR). They can be directly read and written using the *mrs* and *msr* opcodes. Four of the status bits (*Negative*, *Zero*, *Carry*, *Overflow*) are read- and writeable in user mode and can thus immediately be used as predicates[3]. This limits the maximum intraprocedural nesting depth of conditional block to a value of four, an acceptable value for code targeted at time critical systems given that loop constructs do not stress the predicate stack.

Possibilities to support even deeper nesting include extending the predicate stack to also use the status bits defined as unused by the ARM manual (a total of eight flags) and swapping out lower parts of the predicate stack to the program stack.

4 Experimental Evaluation

In order to gain experimental evidence regarding the methodologies outlined in this paper we have made an attempt to reproduce the results from [10] on the hardware platform described in Section 3. In particular we looked at the Bubble Sort algorithm, the benchmarks were compiled with gcc-3.4.5 in order to exercise them on the Gumstix only using minimal bare-metal configuration, restricted to serial I/O drivers and timing infrastructure.

By inspecting the assembly code generated when using

[1]http://www.arm.com/documentation/Instruction_Set/index.html
[2]http://docwiki.gumstix.org/Basix_and_connex
[3]http://www.arm.com/documentation/Instruction_Set/index.html

aggressive ("O3") optimisation we observed, that that the algorithm is not suitable for single-path conversion, because gcc already heavily relies on predicated instructions instead of branches. Further investigations showed that conditional blocks up to about five statements in the C source code are almost always compiled to predicated instructions.

Hence the preliminary conclusion we draw is, that many of the well known sorting algorithms with tight loops and brief conditional blocks are unsuitable for post-pass transformation when compiled with full optimisation using gcc for ARM. We are thus looking to conduct measurements on application code, as it is not always possible to express domain-specific programs as elegantly as the discussed examples. In particular we will be looking at the controller loop of the JAviator quadrotor UAV[4].

5 Conclusion and Future Work

In this work in progress paper we have introduced the notion of a predicate stack and presented a conceptual model for single-path execution of predicated code. Furthermore we have outlined an assembly transformation algorithm that translates arbitrary programs to single-path code. We are aware that unconditional single-path transformation is a brute force approach when applied to domain-specific programs rather than well-behaved and optimised algorithms. Nevertheless studying the behaviour of such programs with regard to single-path execution is an important direction we are setting out for further work. Also our current effort is constrained to intraprocedural transformations, further work is required to look at single-path execution from an intraprocedural point of view. Moreover, we need to collect experience regarding the behaviour of single-path code in the context of full blown embedded systems, rather than isolated benchmarks [12].

Finally we acknowledge that single-path execution is only one among a number of orthogonal issues towards improved WCET analysis and predictability. Software managed caches (often referred to as "scratchpad memory") and fine-grained control over CPU subsystems (like for example I-Cache locking [3]) are posing interesting challenges, all the more when combined with single-path execution, as presented in this paper.

6 Acknowledgements

The author would like to thank Harald Röck for perpetually providing insight regarding ARM assembly intrinsics and Horst Stadler for helping with the experimental evaluations in the course of this effort.

[4]http://javiator.cs.uni-salzburg.at

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *POPL*, pages 177–189, 1983.

[2] B. Decker and D. Kästner. Reconstructing control flow from predicated assembly code. In A. Krall, editor, *SCOPES*, volume 2826 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2003.

[3] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2007. ACM.

[4] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[5] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO*, pages 45–54. ACM/IEEE, 1992.

[6] J. C. H. Park and M. S. Schlansker. *On predicated execution*. Hewlett Packard Laboratories, 1991.

[7] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *RTCSA*, pages 442–. IEEE Computer Society, 1999.

[8] P. Puschner. The single-path approach towards wcet-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pages 699–704, Dec. 2003.

[9] P. P. Puschner. Algorithms for dependable hard real-time systems. In *WORDS*, pages 26–31. IEEE Computer Society, 2003.

[10] P. P. Puschner. Experiments with wcet-oriented programming and the single-path architecture. In *WORDS*, pages 205–210. IEEE Computer Society, 2005.

[11] P. P. Puschner and A. Burns. Writing temporally predictable code. In *WORDS*, pages 85–94. IEEE Computer Society, 2002.

[12] P. P. Puschner and R. Kirner. From time-triggered to time-deterministic real-time systems. In B. Kleinjohann, L. Kleinjohann, R. J. Machado, C. E. Pereira, and P. S. Thiagarajan, editors, *DIPES*, volume 225 of *IFIP*, pages 115–124. Springer, 2006.

[13] H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA*, pages 23–30. IEEE Computer Society, 2000.