An Early Real-Time Checker for Retargetable Compile-Time Analysis

Emilio Wuerges, Luiz C. V. dos Santos, Olinto Furtado
Federal University of Santa Catarina
Computer Science Department
Florianopolis, SC, Brazil
+55-48-3721-7549
{emilio, santos, olinto}@inf.ufsc.br

Sandro Rigo
State University of Campinas
Institute of Computing
Campinas, SP, Brazil
+55-19-3521-5849
sandro@ic.unicamp.br

ABSTRACT

With the demand for energy-efficient embedded computing and the rise of heterogeneous architectures, automatically retargetable techniques are likely to grow in importance. On the one hand, retargetable compilers do not handle real-time constraints properly. On the other hand, conventional worst-case execution time (WCET) approaches are not automatically retargetable: measurement-based methods require time-consuming dynamic characterization of target processors, whereas static program analysis and abstract interpretation are performed in a post-compiling phase, being therefore restricted to the set of supported targets. This paper proposes a retargetable technique to grant early real-time checking (ERTC) capabilities for design space exploration. The technique provides a general (minimum, maximum and exact-delay) timing analysis at compile time. It allows the early detection of inconsistent time-constraint combinations prior to the generation of binary executables, thereby promising higher design productivity. ERTC is a complement to state-of-the-art design flows, which could benefit from early infeasiblity detection and exploration of alternative target processors, before the binary executables are submitted to tight-bound BCET and WCET analyses for the selected target processor.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems (J.7)**]: Real-time and embedded systems

General Terms

Design, Languages, Verification

Keywords

Compile-time WCET analysis, Time-constraint feasibility analysis

1. INTRODUCTION

The huge research effort to obtain tight *worst-case execution time* (WCET) bounds for processors with dynamic features (e.g. cache, out-of-order pipelines, branch prediction) has led to the rise of efficient and provenly correct techniques [12], which are currently available for industrial usage, tailored to a portfolio of popular real-life processors. Such an achievement has solved the important problem of obtaining tight-bound WCET estimates for a given processor, while benefiting from the processor's dynamic features to maximize average performance. However, criteria other than performance and predictability may come into play. For instance, real-time constraints could be met by running the application on a simpler, more energy-efficient processor or even by using scratch-pad memories instead of caches. For applications demanding multi-objective optimization, *design space exploration* (DSE) is mandatory. The designer should be allowed to try with several candidate processors in the early design phases, before selecting a target processor to undertake an elaborate tight-bound WCET analysis. If time-constraint analysis is let as an afterthought, DSE could not benefit from pruning inadequate alternatives and design productivity is likely to be hampered by trial-and-error. Therefore, a tradeoff between speed and estimate tightness should make early real-time checking viable. However, no exploration is practical without proper compiler support for the candidate processors. Those reasons motivated us to develop a retargetable time-constraint analysis technique that allows the early detection of inconsistent time-constraint combinations prior to the generation of binary executables, for the sake of higher design productivity and broader design space exploration.

The proposed technique automatically retargets the micro-architecture-dependent aspects of a compiler's backend so as to grant it early real-time checking (ERTC) capabilities. It handles multiple time constraints of distinct types (maximum, minimum, and exact-delay).

This paper is organized as follows. Section 2 reviews retargetable compilers and WCET analysis. Section 3 shows an example of how our technique works. Sections 4 and 5 formalize program representation, timing modeling and constraint specification. The main underlying algorithm is shown in Section 6. Section 7 reports the experimental results. Our conclusions are drawn in Section 8.

2. RELATED WORK IN BRIEF

The packages built around *architecture description lan-*

guages (ADLs) [4] [8] [7] [1] provide retargetable compiling, assembly, and linkage, as well as the automatic generation of an *instruction-set simulator* (ISS) for the target processor. Therefore, ADL packages can provide instruments for dynamic WCET analysis, but at the expense of running the program on the generated ISS [14], which is unsafe and rather time consuming. To efficiently check time constraints at compile time, without the need of simulating the program, latencies should be inferred from the processor and memory subsystem ADL description and be annotated into the program representation. However, some ADLs (like LISA) do not describe instructions individually (but as a composion of operations), which makes latencies difficult to infer from the target's ADL description.

There are two main approaches for execution time analysis: static and dynamic [3] [12]. In the first, the execution time is calculated from the analysis of the code, which does not need to be executed or simulated. This approach involves a (high-level) control flow analysis and a (low-level) hardware-aware analysis. On the other hand, the dynamic approach is based on measuring the time spent executing the code (or a segment of it), for a set of judiciously-chosen input stimuli, either by running it on the actual target or by simulating its execution on a hardware model. Most of the static tight-bound WCET analysis rely on two key foundations: the *integer linear programming* (ILP) formulation and the *abstract interpretation* (AI) theory [2]. Initially, it was shown that ILP could be used to address path analysis without the need to explicitly enumerate paths [10] and could be extended to model caches and pipeline. Since such extension was likely to lead to high analysis times, further investigation has shown that AI is more adequate for value and cache analysis, while ILP is more suitable for pipeline and path analysis [13]. A combination of ILP and AI was released in the form of a tool for industrial usage [12]. The important problem of computing false paths prior to WCET analysis is addressed in [6], for instance, where AI is employed to avoid visiting basic blocks that are not reached under some execution condition.

For the above reasons, we developed a technique with the following capabilities:

- Automatically retargetable inference of latencies from an ADL description of a processor's microarchitecture and memory subsystem;
- Language-independent specification of different types of real-time requirements (minimum, maximum, and exact-delay constraints);
- Early checking of constraint feasiblity at compile time.
- Lightweight timing modeling (suitable for DSE) to provide preliminary real-time guarantees.
- Complementarity to state-of-the-art tight-bound post-compiling BCET/WCET analysis.

3. AN ILLUSTRATIVE EXAMPLE

Assume an application where I/O operations require a handshaking mechanism for the reading (writing) of memory-mapped I/O registers that store input data from a sensor (or output data for an actuator). Suppose that a process sends a request by invoking the procedure in Figure 1 (where the actual access to the I/O register is omitted for simplicity). Assume that, after the program posts a request, the peripheral must respond with an acknowledgment not more than 12 cycles later (Constraint 1) and that the program must poll the peripheral's acknowledgement flag not earlier than 3 cycles after posting the request (Constraint 2).

Constraints 1 and 2 are specified by means of compiler directives at lines 4, 7, 9, and 12. The first element within a directive specifies whether a starting or an ending point of a given constraint, whereas the second assigns a label to it. A directive representing a starting point has two extra elements: one states the type of constraint (maximum or minimum), the other provides its value (expressed in cycles).

```
 1 void block(unsigned char * req,
 2   unsigned char * ack)
 3 {
 4   asm volatile("begin,constr1,MAX,12;");
 5   /* request activation */
 6   *req = 0x01;
 7   asm volatile("begin,constr2,MIN,3;");
 8   /* here lies some (omitted) code */
 9   asm volatile("end,constr2;");
10   while(!(*ack));
11   /* acknowledgement activated */
12   asm volatile("end,constr1;");
13 }
```

Figure 1: The instrumented source code

The code is then compiled to an intermediate representation (IR), which is annotated with the instruction latencies (inferred from an ADL description of a candidate microarchitecture). Each directive in the source code induces a *pseudo-instruction* in the code's IR for marking the beginning or the ending of Constraints 1 and 2. Those pseudo-instructions will serve as anchors: they will be the starting and ending vertices of paths whose minimum or maximum length will be measured. To check MIN (MAX) constraints, we rely on shortest (longest) path algorithms (see Section 7). Exact-delay checking is obtained by a combination of MIN and MAX constraints (see Section 5.1). For every violated contraint, a diagnosis report returns its type, its number of specified cycles, and the actual number of cycles.

4. PROGRAM REPRESENTATION

A program consists of a set of procedures or functions $P = \{p_1, p_2, \ldots, p_n\}$. Each procedure p_i is modeled by a *control-flow graph* $CFG = (B, F)$, where each vertex $b \in B$ is a *basic block* (BB) and where each edge $f \in F$ represents the flow of execution between basic blocks. Every BB is modeled by a *data-flow graph* $DFG = (V, E)$, where each vertex $v \in V$ represents an instruction and where each edge $(u, v) \in E$ represents a data dependence.

As the code goes through different levels of optimization, program representation is captured in distinct intermediate forms. An *intermediate representation* (IR) is a valid representation of the program where some of the original elements may be omitted since they were already taken into account by previous optimizations. For instance, after instruction scheduling, an IR can represent the instructions as an ordered set, since scheduling guarantees that this order satisfies the precedence relation in the DFG.

Given a basic block $b_i \in B$ and its associated $DFG = (V_i, E_i)$, let I_i be a linearly ordered set obtained from the set V_i such that the partial order E_i is satisfied. The adopted IR on entry to the backend (from now on referred simply as IR) consists of the CFGs representing each procedure and the (ordered) instructions of each BB, as formalized below.

The *intermediate representation* of a program is the tuple $IR = (P, C, I)$, where:

- $P = \{p_k : p_k \text{ is a procedure or function}\}$;
- $C = \{\forall p_k \in P : (B_k, F_k)\}$;
- $I = \{\forall b_i \in B \wedge \forall (B, F) \in C : I_i\}$.

5. MODELING OF TIMING AND CONSTRAINTS

5.1 Capturing real-time requirements

Real-time constraints are captured by means of a pair of scope delimiters, called *source* and *sink*. A delimiter points to an instruction within a BB. Since constraints may be imposed across BBs, a delimiter must identify not only the instruction at which it is pointing, but also the BB owning that instruction. This notion can be formalized as follows.

A *real-time constraint* can be represented as a tuple $tc = (b_i, d_i, b_j, d_j, t, \tau)$, where:

- $b_i \in B$ is the BB that contains the instruction pointed by source;
- $d_i \in Z^+$ is the position in I_i at which the source is pointing;
- $b_j \in B$ is the BB that contains the instruction pointed by sink;
- $d_j \in Z^+$ is the position in I_j at which the sink is pointing;
- $t \in Z^+$ is the specified number of cycles between source and sink;
- $\tau \in$ {MIN, MAX} is the contraint type specifying t as either a minimum or maximum value.

An exact-delay constraint of t cycles is specified by a pair of time constraints $tc1 = (b_i, d_i, b_j, d_j, t, MIN)$ and $tc2 = (b_i, d_i, b_j, d_j, t, MAX)$.

5.2 Capturing micro-architecture timing

The *latency between instructions* u and v, written $\lambda(u, v)$, is the number of intervening cycles necessary to guarantee that the data produced by u is available before it can be consumed by v. Both the pipeline and the memory subsystem may contribute to the overall latency, as follows.

The *pipeline latency between instructions* u and v, written $\lambda_p(u, v)$, is the minimum number of intervening cycles between the time when a value is produced by u on exit from a pipeline stage and the time when that value is available on entry to the pipeline stage where it is consumed by v.

In order to model the contribution of the memory subsystem to the latency, we need to distinguish instruction classes and then quantify the impact of memory accesses. Let $\tau : I \rightarrow \{store, load, reg\}$ be the *type* function, which maps every instruction $u \in I$ to an instruction type $\tau(u)$, where *load* and *store* denote instructions reading and writing operands in memory, respectively, and *reg* denotes instructions whose operands are always in registers.

Memory access time may depend on the actual address value and on the access type, which can be distinguished as follows. Let ϵ_u denote the effective data address referenced by instruction u such as $\tau(u) \neq reg$. Let $R(\epsilon)$ and $W(\epsilon)$ denote the number of cycles spent in reading and writing, respectively, at a given address ϵ. The latency between instructions u and v can be obtained as follows:

$$\lambda(u, v) = \begin{cases} \lambda_p(u,v) & \text{if } \tau(u) = reg \\ R(\epsilon_u) + \lambda_p(u,v) & \text{if } \tau(u) = load \\ W(\epsilon_u) + \lambda_p(u,v) & \text{if } \tau(u) = store \end{cases}$$

Whereas $\lambda_p(u, v)$ is constant for a given pipeline, $R(\epsilon_u)$ and $W(\epsilon_u)$ vary depending on whether the address ϵ_u is mapped to a scratch-pad, a memory-mapped I/O register, or a cache. Since in the last case, $R(\epsilon_u)$ and $W(\epsilon_u)$ vary dynamically, they can not always be inferred from the system description and, therefore, require a model of cache dynamic behavior. Since such a model is expected to return an accurate estimate for the number of cycles spent in memory access, it will be called a *memory oracle*.

Let $\varphi : I \rightarrow Z^+$ be the *issue* function, which maps every instruction $v \in I$ to an issue time step $\varphi(v)$. The *actual number of intervening cycles* between instructions u and v as a result of dynamic issue, written $\varphi_{(u,v)}$, is obtained by $(\varphi(v) - \varphi(u)) - 1)$.

Each instruction may contribute to the execution time with a number of stall cycles in the interval $[0, \lambda(u, v)]$, as a result of a data hazard, depending on how far apart instructions u and v were issued, as formalized below.

Given an instruction v that is data dependent on u, the *number of stall cycles* between u and v, written $\sigma(u, v)$, is given by:

$$\sigma(u, v) = \begin{cases} 0 & \text{if } \varphi_{(u,v)} \geq \lambda(u, v) \\ \lambda(u, v) - \varphi_{(u,v)} & \text{if } \varphi_{(u,v)} < \lambda(u, v) \end{cases}$$

To obtain fast estimates for BCET and WCET, we assume that the control-flow structure is preserved at runtime. This pessimistic assumption has two pragmatic consequences: 1) WCETs can be bound without the need of enumerating execution scenarios; 2) execution times can be measured relatively to BB boundaries, as follows.

To compute the number of cycles spent by a sequence of (overlapped) instructions within a BB, we need to take into account the outcome of code scheduling. Let $\phi : V \rightarrow Z^+$ be the *schedule* function that maps every instruction $v \in V$ to a single time step $\phi(v)$ within the scope of a BB, which determines the cycle when instruction fetching is launched. Let α_v denote the address of instruction v and let $R(\alpha_v)$ denote its fetching delay.

The incremental contribution of a newly launched instruction v to the program's execution time is the number of non-overlapping instruction cycles, which comprises the cycles spent fetching v and, possibly, the cycles spent waiting for data produced by some preceding instruction u. This notion can be formalized as follows.

Given a BB and its DFG (V, E), the *relative latency* up to instruction x within that BB, written λ_x, is obtained as:

$$\lambda_x = \sum_{v=\phi^{-1}(0)}^{x} (R(\alpha_v) + \max_{(u,v)\in E} \sigma(u,v))$$

5.3 Combining with program timing

Up to this point our modeling casts processor and memory properties into the program representation. To perform proper BCET/WCET analyses, we have to account for the effect of loop iterations, as follows. We assume that the compilation steps leading to the IR were constrained to enforce a CFG topology exibiting a *nesting structure*, i.e., given two loops l_i and l_j, only one of the following conditions holds: 1) l_i and l_j are entirely disjoint; 2) l_i is entirely contained within l_j; 3) l_j is entirely contained within l_i. As a result, loop behavior can be captured as follows. Given a BB $b_i \in B$, let L_i be the set of nested loops containing b_i. Let n_l be the number of iterations of a loop $l \in L_i$. Thus the number of invocations N_i of b_i is given by:

$$N_i = \begin{cases} 1 & \text{if } L_i = \emptyset \\ \prod_{l\in L_i} n_l & \text{if } L_i \neq \emptyset \end{cases}$$

We can now combine processor and loop properties. The overall time spent executing the instructions of a BB (possibly contained in a loop nesting) is the time spent in executing the BB instructions once (taking latencies and stall cycles into account as discussed in Section 5.2) multiplied by its number of invocations, as formalized in the following.

Given a BB b_i and its DFG (V_i, E_i), the latency of b_i, written $\lambda(b_i)$, is computed as:

$$\lambda(b_i) = N_i \times \sum_{v\in V_i} (R(\alpha_v) + \max_{(u,v)\in E_i} \sigma(u,v))$$

Now we can generalize this notion to a path in the CFG.

Given a path $\pi = (b_0, b_1, b_2, ..., b_k, ..., b_n, b_{n+1})$ in a CFG, the *latency in-between* b_0 and b_{n+1} through π, is $\lambda(\pi) = \sum_{k=1}^{n} \lambda(b_k)$. (We deliberately excluded the starting and ending vertices from path π in computing the latency for reasons that will be made clear in Section 5.4).

5.4 Calculating actual delays

Assume that the source and sink delimiters of a time constraint point to instructions x and y, respectively. Figure 2 shows the possibilities when both instructions lie within a same BB: (a) corresponds to a constraint inside a BB; (b) captures a constraint across a self-loop. Figure 3 distinguishes two scenarios when instructions x and y belong to different BBs b_i and b_j: (a) captures a constraint through a forward path; (b) depicts a constraint across a cycle in the CFG (i.e. across a loop body in the code). $\lambda(b_i, b_j)$ denotes the latency measured in-between b_i and b_j on some path starting at b_i and ending at b_j.

With the help of Figures 2 and 3, let us now formalize a crucial notion to our analysis. Let b_i and b_j be BBs for which I_i and I_j denote their linearly ordered set of instructions. Given the instructions $x \in I_i$ and $y \in I_j$, which are respectively pointed by the source and sink delimiters of a given time-constraint, the *calculated delay* between x and y, written $\delta(x, y)$, is obtained as follows:

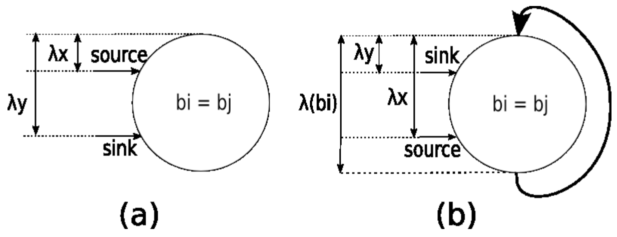

Figure 2: Delimiters point to the same BB

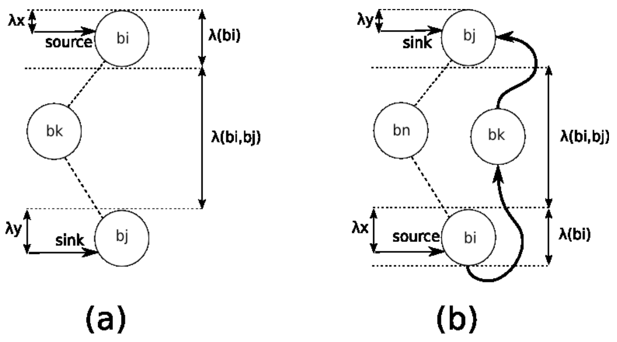

Figure 3: Delimiters point to distinct BBs

$$\delta(x,y) = \begin{cases} \lambda_y - \lambda_x & \text{if } b_i = b_j \wedge \lambda_x \leq \lambda_y \\ \lambda(b_i) + \lambda_y - \lambda_x & \text{if } b_i = b_j \wedge \lambda_x > \lambda_y \\ \lambda(b_i) - \lambda_x + \lambda(b_i, b_j) + \lambda_y & \text{if } b_i \neq b_j \end{cases}$$

Note that the first and second clauses hold, respectively, in the scenarios of Figure 2a and 2b. Observe that the third clause holds for both scenarios in Figure 3.

5.5 Bounding BCET and WCET

Before we can evaluate the minimum and maximum delays between two instructions, we have to properly bound every variable in $\delta(x, y)$. We first bound BB and path latencies and then we discuss how to bound instruction latency.

Let ${n_l}^{max}$ (${n_l}^{min}$) be the maximum (minimum) number of iterations of a loop $l \in L_i$. Let ${N_i}^{max}$ (${N_i}^{min}$) be the overall number of invocations when $n_l = {n_l}^{max}$ ($n_l = {n_l}^{min}$) for every loop $l \in L_i$.

Then we can define the maximum and minimum latencies of a given BB b_i as follows:

$$\lambda^{max}(b_i) = \lambda(b_i) \mid N_i = {N_i}^{max}$$
$$\lambda^{min}(b_i) = \lambda(b_i) \mid N_i = {N_i}^{min}$$

Now we can generalize the bounding to a path in the CFG. Let $b_i \xrightarrow{\pi} b_j$ denote that b_i reaches b_j through path π. Then the *maximum* and *minimum latencies in-between* them are, respectively:

$$\lambda^{max}(b_i, b_j) = max\{\forall\pi \mid b_i \xrightarrow{\pi} b_j : \lambda(\pi)\}$$
$$\lambda^{min}(b_i, b_j) = min\{\forall\pi \mid b_i \xrightarrow{\pi} b_j : \lambda(\pi)\}$$

We can define the *minimum* and *maximum calculated delays* between x and y, respectively, as follows:

$$m(x,y) = \delta(x,y)|((\lambda(b_i)=\lambda^{min}(b_i)) \wedge (\lambda(b_i,b_j)=\lambda^{min}(b_i,b_j)))$$
$$M(x,y) = \delta(x,y)|((\lambda(b_i)=\lambda^{max}(b_i)) \wedge (\lambda(b_i,b_j)=\lambda^{max}(b_i,b_j)))$$

Finally, we discuss how instruction latencies can be bound within BB and paths.

Note that, to obtain λ_x we must determine the value of $\sigma(u,v)$ for every pair of dependent instructions. However, $\sigma(u,v)$ depends on $\varphi(u,v)$, which can not be tightly bounded at compile-time without (explicitly or implicitly) enumerating all distinct execution scenarios induced by dynamic pipeline effects. Since DSE can not afford the computational effort of such enumeration, we have to rely on loose bounds.

Given an arbitrary instruction u, to obtain upper and lower bound estimates for $R(\alpha_u)$, $R(\epsilon_u)$, and $W(\epsilon_u)$, we assume the existence of a memory oracle implementing some state-of-the-art tight-bound cache modeling based upon AI, such as in [13].

Let us denote the upper and lower bound estimates of an arbitrary variable f by $\hat{f}$ and $\check{f}$, respectively. To obtain an upper bound estimate for stall cycles, we pessimistically assume that every pair of dependent instructions was issued as close as possible ($\varphi(u,v) = 0$) so that the whole latency between them is exposed. Under this assumption, we have:

$$\hat{\sigma}(u,v) = \begin{cases} \lambda_p(u,v) & \text{if } \tau(u) = reg \\ \hat{R}(\epsilon_u)+\lambda_p(u,v) & \text{if } \tau(u) = load \\ \hat{W}(\epsilon_u)+\lambda_p(u,v) & \text{if } \tau(u) = store \end{cases}$$

To obtain a lower bound estimate for stall cycles, we optimistically assume that dependent instructions were issued sufficiently far apart so as to hide their latency, i.e. $\check{\sigma}(u,v) = 0$.

Observe that, under those deliberately loose bounds, estimation speed depends essentially on the memory oracle, which can rely on efficient AI, while pipeline scenario enumeration would require ILP (whose higher computational effort would hamper DSE). By using tight-bound analysis at the memory oracle only (and not for bounding pipeline behavior), estimates are likely to be kept within the same order of magnitude as full tight-bound estimates, which is acceptable for DSE purposes.

6. THE PROPOSED REAL-TIME CHECKER

Our checker makes the following assumptions: 1) An ADL description of the target microarchitecture is available from which the latencies can be inferred; 2) Prior compilation steps enforced the CFG topology defined in Section 5.3; 3) The number of invocations of each BB was pre-determined through classical loop analysis; 4) False path analysis [6] was performed beforehand so that latencies of BBs belonging to false paths are not taken into account.

Given a model of the `target` processor and memory subsystem, and the set of procedures `P` of a program, Algorithm 1 finds the set of infeasible time constraints (if any). At line 1, it invokes a function to extract, from the processor model, the latencies of each pair of dependent instructions (function's body omitted for simplicity). Then it checks whether the delay between two instructions fulfils or not the specified value and type for each time constraint within the scope of every procedure. The resulting set of infeasible constraints can then be used either to assert feasibility (if `Infeasible` $= \emptyset$) or to pinpoint the pairs of instructions violating time constraints (if `Infeasible` $\neq \emptyset$).

Algorithm 1 `ERTC(P, target)`

```
1:  annotate-Latencies(P, target)
2:  Infeasible = ∅
3:  for each $p \in P$ do
4:    let $(B, F)$ be the CFG representing $p$
5:    let $TC$ be set of time constraints on procedure $p$
6:    for each tc = $(b_i, d_i, b_j, d_j, t, \tau) \in TC$ do
7:      let $x$ be the instruction at position $d_i$ in BB $b_i \in$ B
8:      let $y$ be the instruction at position $d_j$ in BB $b_j \in$ B
9:      if $(\tau =$ MIN $\wedge$ m$(x,y) < t) \vee (\tau =$ MAX $\wedge$ M$(x,y) > t)$ then
10:       Infeasible= Infeasible ∪ {tc}
11:     end if
12:   end for
13: end for
```

Table 1: Outcome of analysis

Program		PowerPC			SPARC		
[5]		ERTC			ERTC		
	p_i	m(x,y)	M(x,y)	ISS	m(x,y)	M(x,y)	ISS
bs	p_1	6	34	16	9	52	20
ns	p_1	6	53	11	8	89	15
crc	p_1	12	24	18	12	33	21
edn	p_1	11	14	13	13	16	15
edn	p_2	5	37	9	7	41	13
edn	p_3	5	88	9	7	92	13
edn	p_4	15	56	19	16	57	23
edn	p_5	11	249	14	15	270	20
edn	p_6	22	47	30	24	49	34
edn	p_7	12	33	20	14	35	24
edn	p_8	5	19	9	7	21	13
fir	p_1	27	91	33	28	106	37
fdct	p_1	17	483	31	19	493	24
adpcm	p_1	14	14	14	14	14	14
adpcm	p_2	19	36	26	19	36	29
adpcm	p_3	19	35	35	21	45	45
adpcm	p_4	88	159	90	97	168	100
prime	p_1	13	31	30	16	43	43
jfdctint	p_1	16	410	18	21	443	24
fibcall	p_1	11	26	23	13	29	26
insertsort	p_1	28	63	33	31	68	37
janne	p_1	5	32	10	7	46	14

7. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We reused the ISA retargeting of the `llc` compiler [9]. Our ERTC was placed after instruction selection/scheduling and immediatly before code emission. To describe the target microarchitecture, we adopted the ArchC ADL [11].

To check for MIN constraints, we relied on Djikstra's shortest-path algorithm. For verifying MAX constraints, a depth-first search algorithm was used in such a way that backward edges were ignored when calculating longest paths (this allows separate handling of loop iteration and forward execution, reflecting the modeling in Section 5.3).

Since the integration of the memory oracle was not complete by the time of writing, we chose an experimental set-up that assumes constant access time to memory, as if both instruction and data were allocated in scratch-pad memories (i. e., $\forall u \in I : R(\epsilon_u) = W(\epsilon_u) = R(\alpha_u) = 1$).

Table 1 shows the analysis outcome for targets PowerPC and SPARC. For each program, we selected at least one procedure p_i and simultaneously imposed a pair of MAX and MIN constraints delimiting the whole procedure body. Our ERTC detected infeasibility in all expected cases. We also executed all programs in the ISS of each target. Observe that the values obtained from the ISS always fell in-between the values calculated by the ERTC. This is an evidence of the technique's correctness for both targets. Runtimes were measured on a Core2 Duo T5250 processor (1.5GHz, 2MB L2) with 2GB (667 MHz DDR2) of main memory. They varied from 0.09s for `insertsort` (93 lines of code) to 0.65s for `adpcm` (910 lines of code).

8. CONCLUSION AND FUTURE WORK

The experiments have shown the consistency of our analysis. Despite current prototype limitations, the low baseline runtimes indicate that the technique is viable for DSE. Although we could not experiment with more targets so far, we can conclude that any ADL capable of describing instruction timing would allow retargetability as far as it allows efficient inference of latencies. We intend to compare our results to tight-bound BCET/WCET estimates so as to quantify the confidence intervals on which DSE can rely.

9. REFERENCES

[1] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 1999.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[3] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Department of Information Technology, 2002.

[4] A. Fauth, J. V. Praet, and M. Freericks. Describing Instruction Set Processors using nML. In *Proc. European Design and Test Conference (EDTC)*, pages 503–507, 1995.

[5] J. Gustafsson. The Mälardalen WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2007.

[6] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for Infeasible Path Calculation. In *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[7] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proc. Design Automation Conference (DAC)*, pages 510–515, 1998.

[8] J. Ceng et al. C Compiler Retargeting Based on Instruction Semantics Models. In *Proc. Design, Automation and Test in Europe Conference (DATE)*, 2005.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Symp. on Code Generation and Optimization (CGO)*, 2004.

[10] Y. Li, S. Malik, and M. Inc. Performance Analysis of Embedded Software using Implicit Path Enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[11] R. Azevedo et al. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[12] R. Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.

[13] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 144–153, 1998.

[14] X. Li, et al. A Retargetable Software Timing Analyzer Using Architecture Description Language. In *Proc. Asian and South Pacific Design Automation Conference*, pages 396–401, 2007.