Using Trace Scratchpads to Reduce Execution Times in Predictable Real-Time Architectures

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Abstract

Instruction scratchpads have been previously suggested as a way to reduce the worst case execution time (WCET) of hard real-time programs without introducing the analysis issues posed by caches. Trace scratchpads extend this paradigm with support for instruction level parallelism (ILP) while preserving simplicity of WCET analysis. In this paper, we demonstrate trace scratchpads using the MCGREP-2 CPU architecture. We provide a sample algorithm to automatically reduce the WCET of a program using a trace scratchpad, and compare the results with the use of an instruction scratchpad.

We find that the two types of scratchpad are best used together. Instruction scratchpads provide excellent WCET improvements at low cost, but trace scratchpads reduce WCET further by optimizing worst case (WC) paths and exploiting ILP across basic block boundaries. Using our experimental implementation, we have observed WCET improvements over an instruction scratchpad of up to 149% with some Mälardalen WCET benchmarks.

1 Introduction

Instruction caches are widely used within computer systems to compensate for memory access latencies caused by the speed disparity between RAM and CPU. In an embedded *real-time system* (RTS), this causes problems for *worst case execution time* (WCET) analysis [22], and consequently for schedulability analysis [5], because of the difficulty of predicting the dynamic state of the cache as programs run. *Instruction scratchpads* [28] are an architectural solution to this problem, replacing a cache with a static (or program-controlled) memory element that can reduce WCET without complicating timing analysis [20, 29, 33]. However, instruction scratchpads contain only sequential machine code, so *instruction level parallelism* (ILP) in programs can only be exploited by optimizations that are applied dynamically within the CPU. Since dynamic CPU operations are known to create WCET analysis difficulties [13, 16, 34], this places a limit on the WCET reductions possible using only an instruction scratchpad, and poses the question of how the ILP in the machine code can be exploited without complicating analysis.

Trace scratchpads contain explicitly parallel code (*traces*), generated from software by a post compilation step. Our traces are functionally equivalent to (and used in place of) the machine code of the *worst case execution paths* (WC paths [9]) in a program, but the execution time of each path is reduced predictably by parallelisation. We show that the timing effects of traces can be modeled using the *implicit path enumeration technique* for WCET analysis (IPET) [15,23], so that the overall WCET is reduced by a known bound. We are able to apply IPET as described in [23] because basic block execution times are independent of execution history when dynamically adaptive features are replaced by scratchpads. We generate traces from machine code to remove any need to extend a standard C compiler for embedded systems and demonstrate the applicability of our technique to any program independent of the source language. This choice simplifies implementation, but improved results could undoubtedly be obtained by extending a compiler to emit additional information about the code.

The limited space in any scratchpad has to be allocated to program components during or after compilation [27]. There is a general challenge to find space-allocating algorithms to satisfy a goal such as energy usage reduction [14,28] or WCET reduction. Algorithms for the latter problem have been proposed for instruction scratchpads by Wehmeyer and Marwedel [33], Suhendra *et al.* [29] and Puaut and Pais [20]. In [9], Falk *et al.* consider the related problem of allocating space in a locked instruction cache for WCET reduction. However, none of these algorithms are directly applicable to trace scratchpads because each trace has an additional "length" parameter to be selected during allocation, and this has no equivalent in instruction scratchpad or locked cache allocation problems. So we also propose and evaluate a new algorithm to allocate trace scratchpad space.

The parts of this paper are as follows: the next section characterizes trace scratchpads, and section 3 describes our algorithm for allocating trace scratchpad space. We describe how we generate traces in section 4, and describe our test environment in section 5. Section 6 has experimental results, and section 7 has related work.

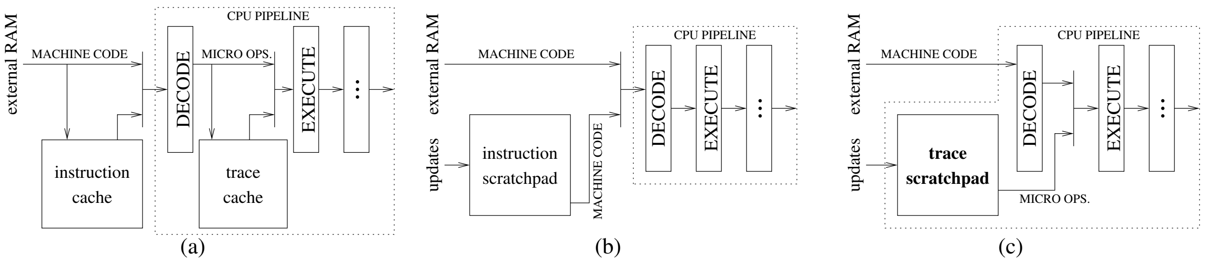


Figure 1. (a) A typical cached architecture, showing possible locations of an instruction cache [18] and a trace cache [25] in relation to CPU components. (b) An instruction scratchpad is a more predictable replacement for an instruction cache [20,29]. (c) A trace scratchpad is part of the CPU pipeline, and can directly control CPU components without decoding. It is also possible to combine scratchpads with caches, for example to reduce energy consumption [14], but this is not considered here.

2 Characterizing a Trace Scratchpad

A trace scratchpad is a high-speed RAM integrated into a CPU pipeline, with size and access latency both similar to a cache or instruction scratchpad [27]. But unlike an instruction cache (Figure 1(a)) or an instruction scratchpad (Figure 1(b)), trace scratchpads store *microinstructions*: each of which may contain a number of distinct *micro-operations*. These directly control CPU functional units without any intermediate decoding stage (Figure 1(c)).

An instruction scratchpad stores conventional machine code, and in many architectures, each machine code instruction specifies just one microoperation, so any ILP must be discovered dynamically by the CPU. In contrast, the microinstructions stored in a trace scratchpad can explicitly issue microoperations to CPU resources in parallel.

An instruction scratchpad is found on the instruction bus and is part of the memory map. Programs can move between execution from RAM and execution from instruction scratchpad using only a jump instruction (Figure 2). But trace scratchpads are not part of the memory map, so a custom instruction is embedded in the program whenever a trace execution should begin. This changes the source of microoperations, from the instruction decoder to the trace scratchpad. Fetching and decoding are suspended while the trace executes. The trace returns control to machine code when it finishes by issuing an “end of trace” microoperation.

Both types of scratchpad have to be explicitly updated by “copy points” in a program [27], unlike trace caches and instruction caches which are dynamically updated by program execution (Figure 3). This can save energy [14], but it also makes scratchpad operation highly predictable because the contents, and consequently the runtime behavior, are precisely known during WCET analysis [20, 29].

2.1 Characterizing a Trace

Instruction scratchpads are easy to update because basic blocks (which contain only sequential machine code) can simply be copied from external RAM to scratchpad.

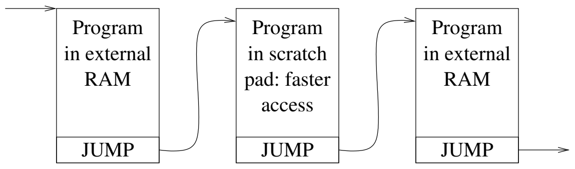


Figure 2. Scratchpad execution paradigm. Instruction scratchpads, trace scratchpads and external RAM hold executable code. Programs can be distributed across all three: execution can jump from any one to any other.

In contrast, trace scratchpads require machine code to be processed by a trace generator.

Traces [12] were first proposed by Fisher [11] and have been widely used to generate *very long instruction word* (VLIW) machine code. A trace includes multiple basic blocks, concatenated together according to a heuristic that conventionally minimizes *average case execution time* (ACET) by making the most likely code path take less time by parallelising operations along that path (Figure 4). The function of the traced code is identical to the original machine code. Conditional branches are treated as assumptions about the most likely code path. These assumptions are evaluated during execution: if they are correct, then trace execution continues. Otherwise a misprediction has occurred and an *exit* is taken. Trace exits ensure that the CPU is in a consistent state for another part of the program. This is done by copying registers and undoing the effects of *speculation* (executing operations before evaluating all of their preconditions). Early trace formation algorithms required complex handling for *side entrances*, where one trace joins another [11], but implementation can be simplified by the use of *superblocks* [6], which are restricted traces with no side entrances.

Traces can be used to reduce WCET by using a formation heuristic that aims to optimize the WC path [9]: this

CONTENTS ARE...		CONTENTS CHANGED... DYNAMICALLY	CONTENTS CHANGED... STATICALLY
	MACHINE CODE	instruction cache	instruction scratchpad
	MICRO OPS.	trace cache	**trace scratchpad**

Figure 3. How trace scratchpads are related to previous work: instruction caches [18], trace caches [25], and instruction scratchpads [20, 29].

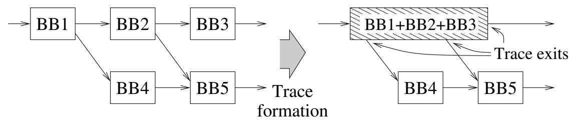


Figure 4. Trace formation example. Execution of WC path [BB1, BB2, BB3] is optimized. The functional correctness of other paths (e.g. [BB1, BB4, BB5]) is preserved.

has been previously applied by Zhao [38] in a compiler optimization context. Janapsatya [14] has previously combined trace formation with instruction scratchpads in order to reduce energy consumption. However, neither approach considers predictable exploitation of ILP or trace selection using an IPET-based methodology (see next section).

3 Issue 1: Allocating Scratchpad Space

A code allocation process must choose a subset of components from a program G for migration to a scratchpad of size C_{max}. Trace scratchpads are not suitable for data storage, so we do not consider the issue of data allocation, and instead recommend the use of an existing scheme for WCET reduction by data scratchpad allocation such as the work of Suhendra *et al.* [29], which can coexist with a trace scratchpad. (Suhendra's approach is applicable to both instruction and data scratchpad allocation.)

Previous work [28, 33] has allocated instruction scratchpad space by using a knapsack problem solving algorithm to trade the benefit of each allocation B_e against the scratchpad space cost C_e for every basic block $b(e)$. However, such knapsack-based approaches assume that allocations will be independent of each other, which is suitable for ACET improvement and energy reduction, but not necessarily for WCET reduction where a new WC path might be introduced by any change to any local execution time cost. In [9], Falk *et al.* classify WCET reduction approaches that do not find new WC paths after each taken decision as *single-path analyses*, and observes that they

```
procedure Find_Candidates($G$):
        $G$ = program as T-graph [23]
        $H$ = max. number of WC paths to find
        $L$ = longest trace to consider
        $W$ = number of starting points to consider per step
        $\Theta = \emptyset$
        $D = []$
        for $i$ from 1 to $H$:
            $Z_0$ = Calculate_WCET($G_\Theta$)  // see section 3.1
            $(V, E) = G$  // members of $E$ represent basic blocks
            Compute_Scores($G$)  // see Figure 6
(1)         sort $E$ in order of $e$.score
            for $j$ from 1 to $W$:
                repeat:
                    $e$ = pop($E$)  // get "highest scoring" block in $E$
                until $e$ is a valid start point for a trace
(2)             $P_e$ = Find_WC_Path($G, e, 0, L$)  // see Figure 7
(3)             $T_e$ = Generate_Trace($P_e$)  // see section 4
                $Z_e$ = Calculate_WCET($G_{\Theta \cup \{T_e\}}$)
(4)             $B_e = Z_0 - Z_e$  // find benefit of this trace
            end for
(5)         find $e$ such that $B_e$ is maximized
            for $k$ from 1 to $L$:
                $P_{e,k}$ = Find_WC_Path($G, e, 0, k$)
                $T_{e,k}$ = Generate_Trace($P_{e,k}$)
                $Z_{e,k}$ = Calculate_WCET($G_{\Theta \cup \{T_{e,k}\}}$)
(6)             $C_{e,k}$ = space cost of trace $T_{e,k}$
                $B_{e,k} = Z_0 - Z_{e,k}$  // benefit of this trace
                if $B_{e,k} > B_e$ then:
(7)                 $B_e = B_{e,k}$
                    $T_e = T_{e,k}$
                end if
            end for
(8)         $\Theta = \Theta \cup \{T_e\}$
            $D = D + [e]$
        end for
end procedure
```

Figure 5. The *Find_Candidates* procedure finds and evaluates candidate traces.

are not optimal. (Similar observations appear in [20, 29].) So we repeat the WCET analysis after each allocation.

Allocating trace scratchpad space for WCET reduction is even more difficult than either locked instruction cache allocation (considered by Falk [9]) or scratchpad allocation (considered by Suhendra [29]) because traces are parameterised by both starting point and length. (We consider the length of a trace to be the number of branch instructions that it includes.) Increasing length may lead to further WCET reductions, but more scratchpad space is required. So the allocation process needs to (1) select good trace starting points, and (2) select good lengths for each trace, and also (3) account for possible dependences between traces. Our algorithm is as follows:

1. Find *candidate traces* using the procedure shown in Figure 5. This repeatedly: (1) identifies the basic blocks that currently make the greatest contribution to WCET us-

```
procedure $Compute\_Scores(G)$:
    $(V, E) = G$
    for $e \in E$:
        $P_e$ = $Find\_WC\_Path(G, e, e, \infty)$
        $e$.dom_cost = $\sum_{x \in P_e} \gamma(x)$
    end for
    for $e \in E$:
        $P_e$ = $Find\_WC\_Path(G, e, e, \infty)$
        $e$.score = $\sum_{x \in P_e} x$.dom_cost $\times f(x)$
    end for
end procedure
```

Figure 6. Heuristic to evaluate the score of each basic block: this is an indication of its contribution to WCET.

```
function $Find\_WC\_Path(G, e_1, e_n, l)$:
    if $l \neq 0 \wedge e_1 \neq e_n$:
        $(V, E) = G$
        $(v_0, v_1) = e_1$
        if $|\{v_2 : (v_1, v_2) \in E\}| > 1$:
            // $e_1$ is followed by a branch
            $l = l - 1$
        end if
        for $(v_1, v_2) \in E$:
            if $f((v_1, v_2)) > \frac{1}{2} f(e_1)$:
                return $[e_1]+$ $Find\_WC\_Path(G, (v_1, v_2), e_n, l)$
            end if
        end for
    end if
    return $[e_1]$
end function
```

Figure 7. Heuristic to find a subsequence of basic blocks within the WC path, beginning at basic block e_1 and ending at either e_n or after l branch points.

ing the score heuristic (Figure 6), then (2) finds the WC paths starting at W of these using the *Find_WC_Path* function (Figure 7), assuming that the path length l is maximized ($l = L$). (3) A trace is generated for each of these paths, and (4) the WCET reduction benefit of that trace is computed. (5) The algorithm chooses the trace starting point e with the greatest benefit B_e, then (6) evaluates the cost and benefit of traces with lengths from 1 to L beginning at e. The trace with the highest benefit is found (7), then added to the program (8). This final step ensures that subsequent iterations will take representatives of the chosen traces into account, enabling new WC paths to be found once earlier ones have been eliminated. However, these representatives do not necessarily become part of the final allocation.

2. Solve an knapsack-like problem to calculate which of the candidate traces $T_{e,k}$ should be selected in order to maximize the total benefit, while keeping the total space cost within the scratchpad size limit. This is similar to a pure knapsack algorithm, as previously applied for scratchpad allocation [27, 33], but it is extended by two new constraints: (1) traces of different lengths k and a common start point e are mutually exclusive, and (2) traces can only be allocated in the order they were added to Θ, to ensure that WC paths are always allocated in order of their effect on WCET.

3. Finally, evaluate the WCET of the program with the selected traces.

The algorithm carries out $HW + HL + 1$ WCET computations by IPET, each of which involves finding a solution to a integer linear program. This is NP-hard in general. However, Li and Malik state that "if we restrict our functionality constraints to those that correspond to the constructs in the IDL language [proposed by Park [19]], then the... problem is equivalent to a network flow problem, which can be solved in polynomial time" [15]. Therefore, although exponential time may be required to run the algorithm in some cases, practical programs can avoid this by limiting the constraints that are used.

Our algorithm relies on some heuristic assumptions to

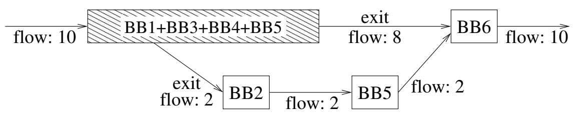


Figure 8. A trace. BB1, BB3, and BB4 are always executed as part of the trace, but BB2 is reached after a trace exit. Consequently, BB5 may be executed either as part of the trace or as machine code. IPET constraints account for this.

cut down the search space: (1) that the best trace starting points can be identified by looking at the WCET contributions of each basic block; (2) that it is sufficient to consider only a limited number (W, L or H) of "most likely" choices at each decision point using the score heuristic; and (3) that a trace at starting point e with length $l \leq L$ can be represented in the IPET model by a trace with the same starting point e and length L. However, these assumptions only affect the quality of the final allocation. The safety of the WCET computation is dependent only on the correctness of our IPET model, because the effects of each trace are always checked using an IPET computation.

3.1 Trace IPET Model

Traces need to be incorporated into a WCET analysis model so that their effects on WC paths can be computed. This modeling is more complex than instruction scratchpad modeling, where only the execution cost $\gamma(e)$ of an individual basic block $b(e)$ needs to be adjusted to evaluate the effect of moving that block into scratchpad. The extra complexity comes from control flow surrounding each trace: an example is shown in Figure 8, where BB5 may be executed outside the trace (as machine code) or as part of the trace, and therefore has two different execution time costs.

The IPET approach for WCET analysis [15, 23] provides an elegant way to model the maximum execution time of general programs, and also identifies WC paths. It can be extended to model the effects of traces by introducing new problem constraints for each traced basic block. (The number of new constraints introduced by a trace is $O(n)$ in the trace length.)

In [23], Puschner and Schedl describe IPET using a *timing graph* (T-graph) $G = (V, E)$ to represent program structure. The approach is summarized as follows. Each edge of the T-graph $x \in E$ represents a basic block $b(x)$. $\gamma(x)$ is a constant representing the execution time cost of $b(x)$, and $f(x)$ is a non-negative integer variable representing the *worst-case flow* (WC flow) through that basic block. The WC flow is the number of times that basic block is executed in the worst case. Each $f(x)$ is computed by solving an integer linear program to maximize the WCET $Z_G = \sum_{x \in E} \gamma(x) f(x)$ for a T-graph $G = (V, E)$. $f(x)$ values are restricted by three types of constraint, which are expressed as linear equations. These are: (1) conservation of flow at each T-graph vertex $v \in V$, (2) behavioral constraints to describe the program, such as loop bounds and infeasible path information, and (3) relative capacity constraints, which enforce conservation of flow for cycles in the T-graph.

This can be extended to allow traces to be modeled as follows. Let Θ represent a set of traces for a program $G = (V, E)$. We wish to determine Z_{G_Θ}: the WCET of the program plus every trace $T_e \in \Theta$. A trace is considered to be a set of paths $P_{e,i} \in T_e$. Each path $P_{e,i}$ is a sequence of edges $x \in E$.

Let $c(x)$ represent the set of all *contexts* in which the functionality of a basic block $b(x)$ is implemented. This may include at most one machine code context $u(x)$, and any number of traced contexts $t(P_{e,i}, j)$. (Traced contexts are identified as the j-th member of a path $P_{e,i}$.) The purpose of this distinction is to allow WC flow to be considered separately for machine code execution (i.e. $f(u(x))$) and for execution in any path (i.e. $f(t(P_{e,i}, j))$). In effect, a new flow variable is created for each $x \in E$ and each context in $y \in c(x)$. Formally, $c(x)$ is defined as follows:

$$\begin{aligned} \forall x \in E, c(x) = \quad & \{u(x)\} \cup \{t(P_{e,i}, j) : T_e \in \Theta \\ & \wedge\ P_{e,i} \in T_e\ \wedge\ x = P_{e,i,j}\} \end{aligned} \tag{1}$$

In the Puschner and Schedl IPET model, the total worst-case flow through x is defined as $f(x)$. This definition is retained, but now the WC flow through traces and machine code is separated into distinct symbols within $c(x)$. The relation between $f(x)$ and $c(x)$ is:

$$\forall x \in E, f(x) = \sum_{y \in c(x)} f(y) \tag{2}$$

If $c(x) = \{u(x)\}$, then $f(u(x)) = f(x)$: i.e. edges that are not represented in traces are unaffected by traces. Relationships also need to be defined between new flow variables. Flow within a trace is conserved:

$$\forall \alpha, \forall \beta, f(t(P_{e,i}, \alpha)) = f(t(P_{e,i}, \beta)) \tag{3}$$

Flow can only enter a trace path $P_{e,i}$ through the entrance e, where machine code flow is converted into traced flow:

$$\begin{aligned} \forall (v_1, v_2) \in E, \sum_i f(t(P_{(v_1,v_2),i}, 1)) = & \\ \sum_{(v_0,v_1) \in E} f(u((v_0, v_1))) & \end{aligned} \tag{4}$$

Flow can only leave a trace path $P_{e,i}$ through its exit, which is the final member of the sequence, $P_{e,i,|P_{e,i}|}$. Flow then returns to machine code:

$$\begin{aligned} \forall (v_0, v_1) \in E, \sum_i f(t(P_{(v_0,v_1),i}, |P_{(v_0,v_1),i}|)) = & \\ \sum_{(v_1,v_2) \in E} f(u((v_1, v_2))) & \end{aligned} \tag{5}$$

Additionally, a trace T_e always replaces machine code execution in the entrance edge e:

$$\forall e \in E, \exists T_e \implies f(u(e)) = 0 \tag{6}$$

As in the original Puschner and Schedl model, flow is also conserved within machine code execution:

$$\sum_{(v_0,v_1) \in E} f(u((v_0, v_1))) = \sum_{(v_1,v_2) \in E} f(u((v_1, v_2))) \tag{7}$$

Equations 1 through 7 define the new flow variables required to represent any set of traces. However, the execution time cost of each path is still missing. The execution cost of machine code $\gamma(u(x))$ can be calculated as for conventional IPET:

$$\forall e \in E, \gamma(u(x)) = \gamma(x) \tag{8}$$

Each $\gamma(t(P_{e,i}, t))$ can be derived from the total execution time cost of the path $\gamma(P_{e,i})$, which is produced as a side effect of trace generation:

$$\gamma(P_{e,i}) = \sum_j \gamma(t(P_{e,i}), j) \tag{9}$$

It is possible to derive each $\gamma(t(P_{e,i}), j)$ using a series of simultaneous equations: one for each path in T_e. There may be many possible solutions to these equations, but all of them are equivalent, because only the *total* execution time cost of each path is important. Costs can be distributed to edges in any arrangement that satisfies this. The only other constraint on each γ value is that no cost can be negative, i.e.: $\forall \alpha, \gamma(\alpha) \geq 0$.

Other equations such as relative capacity constraints and behavioral constraints are added as for IPET. Since

```
#define SORT_SIZE 100
void Benchmark ( void ) { /* BB0 */
    int     i , swapped ;

    do { /* BB1, BB2 */
        swapped = 0 ;
        for ( i = 0 ; i < ( SORT_SIZE - 1 ) ; i ++ ) {
            int     si0 = to_sort [ i ] ; /* BB3 */
            int     si1 = to_sort [ i + 1 ] ;

            if ( si0 > si1 ) {
                to_sort [ i ] = si1 ; /* BB4 */
                to_sort [ i + 1 ] = si0 ;
                swapped = 1 ;
            } /* BB5 */
        } /* BB6 */
    } while ( swapped ) ; /* BB7, BB8 */
}
```

Figure 10. Bubble sort C source.

these apply to $f(x)$ values, they are incorporated into the model via equation 2. The new WCET is defined as:

$$Z_{G_\Theta} = \sum_{x \in E} \sum_{y \in c(x)} f(y)\gamma(y) \qquad (10)$$

This definition is equivalent to Puschner and Schedl's definition for Z_G when $\Theta = \emptyset$. In the extended model, WCET is computed by maximizing Z_{G_Θ} with an integer linear program solver.

Most importantly, Puschner and Schedl's proof of correctness also applies to the extended model. This is because traces do not introduce new execution paths to the program. They just lower the costs of existing ones. Given all behavioral constraints, Puschner and Schedl showed that: (1) if there is exactly one path through a program G defined as $P_1 = (x_{1,1}, ..., x_{1,n})$, then Z_G can be correctly computed; and (2) if there are n paths $\{P_1, ..., P_n\}$ and Z_G can be correctly computed, then adding a new path P_{n+1} preserves the correctness of Z_G. This still applies if traces are being modeled because the set $\{P_1, ..., P_n\}$ is the same regardless of Θ. Equations 1 through 10 provide the extra constraints and execution times required to describe trace behavior and fulfill the preconditions of Puschner and Schedl's proof.

3.2 Example

Consider the C program in Figure 10. Compiling this program for a RISC-like architecture results in the basic block graph shown in Figure 9. The time cost of each basic block $\gamma(x)$ is measured by execution: these are invariants because dynamically adaptive CPU features are replaced by a trace scratchpad. A constraint set {BB6 $\leq$ 100, BB5 $\leq$ 9900, BB4 $\leq$ 5000} is added to represent worst-case behaviour.

The algorithm shown in Figure 5 is executed. The first algorithm iteration sets $h = 1$. Using IPET, the original WCET is calculated as $Z_{G_\emptyset}$ = 1323607 clock cycles.

The next stage of the algorithm adds candidate traces to the program based on the score heuristic (Figure 6). The scores for $\Theta = \emptyset$ are shown in Table 1. The algorithm evaluates W traces of length L in descending order

Basic block $b(x)$	Exec. cost $\gamma(x)$	WC flow $f(x)$ ($\Theta = \emptyset$)	Score (Figure 6)
BB3	81	9900	1.54e+06
BB5	33	9900	1.12e+06
BB4	37	5000	7.50e+05

Table 1. Possible trace starting points for Figure 10, sorted by Score. Costs are given in clock cycles, obtained using the MCGREP-2 simulator (section 5).

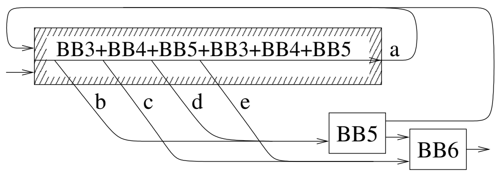


Figure 11. Trace with length 4 and start point BB3 for the example program (Figure 10).

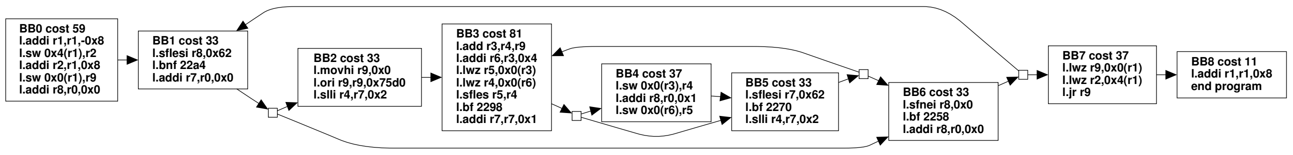
of score, beginning with BB3. For each trace, this evaluation involves *Find_WC_Path* (Figure 7), followed by *Generate_Trace*, followed by WCET calculation. For the purposes of this example, the parameters are $W = H = 3$ and $L = 4$.

Given these settings, the WC path from BB3 of length $l = L = 4$ is [BB3, BB4, BB5, BB3, BB4, BB5]. The trace that is generated for that path is shown in Figure 11. The generator also produces execution time costs for each possible path through the trace (Table 2), and finds the total space cost of the trace (83 scratchpad lines). The WCET is then calculated: it is reduced to 732707, so the trace benefit is 590900.

The algorithm then proceeds as described earlier. It also evaluates traces beginning at BB5 (benefit 990100) and BB4 (benefit 184800). Of these, BB5 gives the highest benefit, so it is chosen as a candidate and reevaluated for each length $l \in [1, L]$ (Table 3). The trace beginning at BB5 with length 4 is added to Θ, as it provides the best benefit with the lowest cost. The algorithm recomputes each score in the presence of this trace, then continues with

Exit Point	Exec. Cost
a	$\gamma(P_{BB3,a}) = 37$
b	$\gamma(P_{BB3,b}) = 40$
c	$\gamma(P_{BB3,c}) = 48$
d	$\gamma(P_{BB3,d}) = 54$
e	$\gamma(P_{BB3,e}) = 61$

Table 2. Execution costs (clock cycles) for each path through Figure 11.



```
BB0 cost 59
l.addi r1,r1,-0x8
l.sw 0x4(r1),r2
l.addi r2,r1,0x8
l.sw 0x0(r1),r9
l.addi r8,r0,0x0

BB1 cost 33
l.sflesi r8,0x62
l.bnf 22a4
l.addi r7,r0,0x0

BB2 cost 33
l.movhi r9,0x0
l.ori r9,r9,0x75d0
l.slli r4,r7,0x2

BB3 cost 81
l.add r3,r4,r9
l.addi r6,r3,0x4
l.lwz r5,0x0(r3)
l.lwz r4,0x0(r6)
l.sfles r5,r4
l.bf 2298
l.addi r7,r7,0x1

BB4 cost 37
l.sw 0x0(r3),r4
l.addi r8,r0,0x1
l.sw 0x0(r6),r5

BB5 cost 33
l.sflesi r7,0x62
l.bf 2270
l.slli r4,r7,0x2

BB6 cost 33
l.sfnei r8,0x0
l.bf 2258
l.addi r8,r0,0x0

BB7 cost 37
l.lwz r9,0x0(r1)
l.lwz r2,0x4(r1)
l.jr r9

BB8 cost 11
l.addi r1,r1,0x8
end program
```

Figure 9. Basic block graph for Figure 10.

Length l	WCET Reduction
0	29700
1	830900
2	661800
3	990100

Table 3. WCET reductions obtained with different trace lengths beginning at BB5.

$h = 2$.

Other traces are evaluated. In the second iteration, start points BB1 (benefit 6100), BB2 (benefit 2900) and BB3 (benefit 4900) are tested. The best is BB1, with length 2. In the third iteration, start point BB6 (benefit 300) is tested. The best is BB6, with length 1. Because this example is very small, there is space for all three traces in the scratchpad. For larger programs, solving the extended knapsack problem would eliminate traces with marginal benefits. In this case, all three can be included. The resulting WCET is computed as 322807. Further reductions are possible by increasing L: for example, setting $L = 20$ widens the search and results in a WCET of 190639. However, adjusting H and W makes very little difference to this small program because practically all WC execution occurs in the [BB3, BB4, BB5] inner loop. There are no significant WC paths outside that loop.

4 Issue 2: Making Traces

The allocation algorithm requires space and time costs for each trace model, and our experimental implementation requires traces to be generated for performance measurement. Therefore, we have implemented a trace generator based on superblocks as described by Chang *et al.* [6]: these traces have only one entry point [12]. Our superblocks differ from previous work in the following ways:

- The source for the trace generator is machine code. This avoids the need to extend a standard C compiler and allows the approach to be applied to any code, including programs not written in C, unstructured programs and closed-source libraries. This choice simplifies implementation, but better results could be obtained by a specialist compiler. For example, a VLIW compiler [12] has access to all data about a program (e.g. the abstract syntax tree), and may use any of this information to generate more efficient superblocks.

- Our environment effectively has two *instruction set architectures* (ISAs): (1) microinstructions from a trace scratchpad, and (2) machine code from an instruction scratchpad or RAM. Conventional VLIW CPUs have only one ISA, and thus the execution of one trace always leads to the execution of another. But that approach is only possible through the use of an instruction cache. Therefore, our trace exits only lead directly to a trace entrance when a trace is being restarted (for example, within a loop). Otherwise, trace exits return to machine code execution. In this way, each trace is independent of all others, and any code can be supported by machine code execution if a suitable trace is not present.

- Our traces are formed along WC paths (see Figure 7). In contrast, VLIW compilers usually try to minimize ACET, and therefore aim for the most likely path. WC path-based trace formation has been previously applied by Zhao [38] within a compiler framework.

Any superblock formation algorithm could be used to build traces, but as the input is machine code, we were led towards a particular choice. We based our experiments on Sohi's out of order superscalar issue algorithm [26], which has previously been implemented as a dynamic CPU component (e.g. in the Simplescalar simulator [4]). Sohi's algorithm is designed to work from machine code and offers an elegant approach for handling exceptions and branch mispredictions, which we use to generate trace exits.

We believe that the low-level implementation details of our modifications to Sohi's algorithm are unimportant as any trace generating algorithm could be used (e.g. [6, 11]). However, high-level changes such as the following will be required:

1. The microoperations provided as input for trace formation, *e.g.* the *register update unit* (RUU, [26]) come from a software queue rather than hardware (cache or RAM).

2. For the main part of each trace, the queue is filled by selecting sequential machine code along the WC path. Branch operations are also placed in the queue, but each is remapped to generate an exception in non-WC conditions: a trace exit to handle a misprediction. In contrast, Sohi's algorithm assumes that branches always follow average case paths.

3. On each iteration, Sohi's algorithm produces at most one microoperation to be executed by each functional unit. This information is encoded as a microinstruction and added to the trace. A more conventional implementation of the algorithm would execute the microoperations immediately.

4. Exceptions are handled by copying the state of the trace generator at each point where an exception might occur, then processing the exception using the copy. This generates an exit handler for the trace, which is placed after the main part of the trace within the scratchpad. The exit handler is reached by a branch microoperation that is embedded in the main part of the trace after copying.

5. Dynamic memory disambiguation is omitted.

6. Code is added to evaluate time and space costs. The total space cost is the number of generated microinstructions. There is a different execution time cost for each exit, based on the number of microinstructions executed on that path. Time costs are added to the T-graph.

5 Evaluation Environment

In order to obtain measurements of trace performance from sample programs, we require a CPU with an integrated trace scratchpad. Unfortunately, adding a trace to an existing CPU design is not easy. We would need to (1) expose the CPU internals so that they can be operated by the trace scratchpad in addition to the existing control unit, and (2) write a program to encode microoperations for the scratchpad memory. Although such changes are possible in principle for any CPU, few CPU designs enable them.

However, we have previously built the MCGREP CPU [35] in which the microoperation layer is (a) extensible, and (b) exposed for external programming through a C *application program interface* (API). The current version, MCGREP-2, is a configurable soft core for *field programmable gate array* (FPGA) devices (Figure 12) with a cycle accurate simulator [36].

The MCGREP-2 CPU can execute conventional machine code from external RAM or an instruction scratchpad, and basic block execution times are independent of execution history. Custom instructions embedded in machine code cause the CPU to stop interpreting machine code and begin executing a trace from scratchpad. (Machine code is interpreted by a *microprogram* that occupies a reserved area of the trace scratchpad.) History dependent timing effects caused by pipeline interleaving are avoided by using a three stage pipeline and a single-slot delayed branch scheme.

The machine code understood by MCGREP-2 is the OpenRISC ORBIS32 ISA, described in [7]. This ISA is also the input code for our trace formation algorithm (section 4). This enables us to build programs for a regular OpenRISC CPU with the unmodified OpenRISC `gcc`

Memory Type	Latency
Inst. Scratchpad	1
Inst. External RAM	10
Data Scratchpad	1

Table 4. Memory timings for evaluation model. Note: a trace scratchpad is integrated into the CPU pipeline, and therefore has no effective latency.

compiler, and then apply scratchpad allocation as a post compilation step: this approach is flexible and demonstrates the compiler-independent nature of our work. The disadvantage is that our traces are limited by the ORBIS32 ISA: integrating trace generation into the compiler would improve our results.

6 Evaluation: Comparison with Instruction Scratchpads

In this section, we compare the WCET reductions available using different scratchpad configurations, using both computed WCETs (from IPET) and measured execution times from our MCGREP-2 implementation. We assume the memory timings shown in Table 4: the data scratchpad is assumed to be large enough to hold all data. In practice, a technique such as [29] could be used to allocate the most important data elements to the scratchpad with similar effect.

We also assume that instruction scratchpad space is allocated using a modification to the algorithm in section 3 that substitutes "basic blocks" for "traces". This restriction sets $L = 1$ since basic blocks do not have a length. (We also set $H = 1000$ so that the instruction scratchpad can be completely filled. None of the benchmark programs include more than 1000 basic blocks, but some are large enough to fill the instruction scratchpad.) With these parameter settings, our algorithm is very similar to the "greedy heuristic" for instruction scratchpad allocation described previously in [29].

For trace scratchpad allocation, the constants are set as shown in Table 5. Higher values of these constants increase the size of the search space to be evaluated. Better results may be obtained, but the search takes longer and the degree of possible improvement diminishes There is no point increasing H and L indefinitely since both are limited by the scratchpad size C_{max}. We set these values to 10 on the grounds that increases beyond 10 do not produce an improvement. Increasing W results in more start points being considered. We have found that the score heuristic is always able to identify the best start point within a window of 5 possibilities for the programs considered. Thus, the settings provide a sufficient search depth for the experiment.

For each of our test programs (Table 6), we calculated the WCET in four different configurations (Table 7) us-

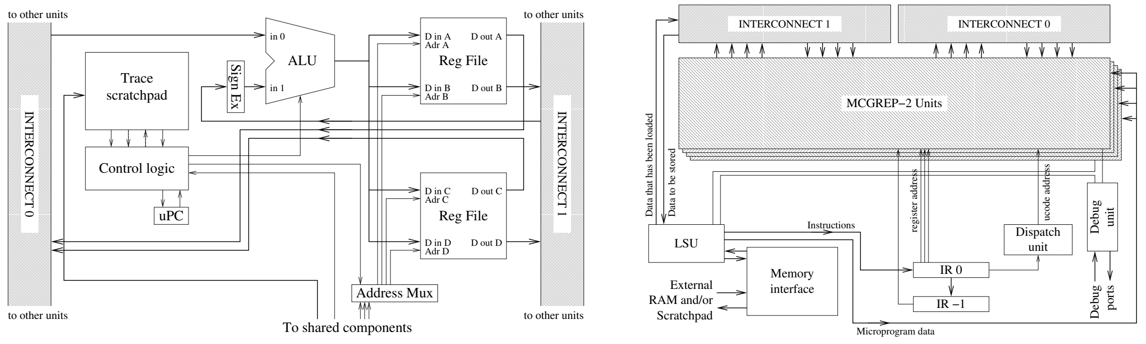


Figure 12. MCGREP-2 CPU: one array unit (left) and top level (right). MCGREP-2 is a type of coarse grained reconfigurable architecture (CGRA) in which each array unit is a small CPU capable of executing code from a trace scratchpad.

Constant	Value
H	10
L	10
W	5

Table 5. Constants used in evaluation. These constants control the search depth (see Figures 5 to 7).

ing IPET. The results are shown in Figure 13 and Table 8. An MCGREP-2 CPU with two functional units is used. For the hybrid configurations, instruction scratchpad space was allocated first. These results illustrate that trace scratchpads should be used in combination with instruction scratchpads. This is because machine code is far more *dense* than microinstructions in terms of operations per memory bit, so more of the program can be present in the scratchpad. However, the results also show that trace scratchpads can reduce WCET beyond what is possible with an instruction scratchpad alone.

WCET reductions using a trace scratchpad are dependent on (1) the ILP available in the WC paths in each program, (2) the trace generator, (3) the CPU microarchitecture, and (4) available scratchpad space. We cannot change the amount of ILP available in each program, and nor do we have another trace generator, but we *can* change the MCGREP-2 CPU architecture and the scratchpad size. Figure 14 shows the mean computed WCET reduction achieved by the use of combined scratchpads instead of an instruction scratchpad, across all programs, and for various scratchpad sizes and CPU configurations. These results indicate that greater WCET reductions are possible by increasing trace scratchpad size, and by increasing the amount of parallelism available within the CPU. Using 64k scratchpads and a 3 unit MCGREP-2 CPU, a WCET reduction of 149% over an instruction scratchpad is achieved for fdct. However, further improvements are limited by the ILP in software [31].

Name	Description
`bs`	Binary search
`bubble`	Bubble sort
`cnt`	Counts non-negative matrix cells
`compress`	Compression program
`crc`	Evaluates a CRC
`div`	Software division
`duff`	Copies an array using Duff's Device
`edn`	Finite Impulse Response filter (1)
`expint`	Computes exponential integral
`fdct`	Fast discrete cosine transform (DCT)
`fibcall`	Fibonacci calculation
`fir`	Finite Impluse Response filter (2)
`insertsort`	Insertion sort
`janne_complex`	Nested loops
`jfdctint`	DCT on 8x8 pixel block
`matmult`	Multiplication of 20x20 matrices
`ndes`	Encryption program
`ns`	Multi-dimensional array search

Table 6. Test programs used in evaluation. All except `bubble` and `div` are from the Mälardalen WCET benchmark set [17]. Where possible, we used the loop constraints provided with the benchmarks.

It is important that WCET estimates are tight (close to the true WCET) to facilitate schedulability analysis [5]. To test this, we measured the execution time of each program in the configurations listed in Table 7 using the MCGREP-2 simulator [36]. The rightmost column of Table 8 shows the results. The values show that very tight (close to 1.0) WCET estimates are possible with trace scratchpads, although as in any WCET analysis approach, the estimates depend on the quality of the behavioral constraints.

7 Related Work

CPUs that dynamically adapt to programs in order to improve ACET have posed significant challenges for

Program	Trace Sp. Bits Used	WC Exec. in Trace Sp. (%)	Inst Sp. Bits Used	WC Exec. in Inst. Sp. (%)	Trace WC Reduction (%)	WCET Tightness
`bs`	8322	19.2	1472	80.8	6.0	0.978
`bubble`	12426	89.0	992	11.0	52.0	0.996
`cnt`	15960	55.2	5952	44.8	53.9	0.856
`compress`	15504	40.7	16320	29.5	14.5	0.948
`crc`	14706	55.5	5376	44.5	15.4	0.912
`div`	15960	64.1	4192	35.9	33.1	0.862
`duff`	14364	63.9	1568	36.1	36.1	1.000
`edn`	15390	60.7	16160	39.3	55.4	0.998
`expint`	15960	56.2	6400	43.8	64.4	0.199
`fdct`	15732	31.4	7968	68.6	39.5	1.000
`fibcall`	9234	72.7	768	27.3	9.0	0.914
`fir`	16188	55.7	6048	44.3	30.3	0.768
`insertsort`	15960	79.0	2016	21.0	55.1	0.778
`janne_complex`	10830	39.2	1152	60.8	6.6	0.960
`jfdctint`	15960	53.2	10272	46.8	52.0	0.852
`matmult`	16302	33.8	6976	66.2	45.2	0.839
`ndes`	13338	16.1	16096	72.2	4.8	0.954
`ns`	12996	52.8	1792	47.2	60.9	0.939

Table 8. Scratchpad space usage and the proportion of WCET spent in scratchpads for the combined configuration. Trace WC reduction is the difference between an instruction scratchpad alone, and instruction and trace scratchpads together (a hybrid configuration). Tightness is the measured execution time divided by the WCET for the hybrid configuration.

Config.	Trace Sp. Size		Inst. Sp. Size	
	Lines	Bits	Words	Bits
RAM only	0	0	0	0
Inst. Sp.	0	0	512	16384
Trace Sp.	143	16302	0	0
Hybrid (both Sp.)	143	16302	512	16384

Table 7. Scratchpad configurations for Figure 13 and Table 8.

WCET analysis because execution times may depend on execution history. Previous work has created effective models for caches [2, 10, 18], which are able to make guarantees about dynamic cache state based on knowledge about execution flow, and thus produce more accurate WCET estimates. But these are not applicable to every practical cache design (e.g. pseudo-LRU and random replacement caches [13]) and must be redesigned for every CPU in order to account for interactions with other CPU components [13], which may themselves depend on execution history. For example, dynamic branch predictors and some pipelines can also introduce a history dependence [16].

The difficulty of applying IPET to a CPU with such dynamic adaptation has been previously noted by Wilhelm [37], citing as an example the increase in problem complexity when caches are modeled as in [15]. This motivates combining the low-level modeling capabilities of *abstract interpretation* (AI) approaches with constraints-based reasoning about high-level flow. Heckmann *et al.* [13] give examples of the application of this combined approach to analyze the behavior of some CPUs with dynamic components. Unfortunately, *some* CPU behaviors cannot be effectively modeled due to the possibility of timing anomalies, as identified by Lundqvist and Stenström [16] and subsequently described by Wenzel *et al.* [34].

These issues have inspired research into new analysis approaches, such as probabilistic modeling as described by Bernat *et al.* [3], and into predictable architectures that are simpler to model [8]. Some approaches take a complex CPU and attempt to simplify some parts of it. One example is VISA, proposed by Anantaraman *et al.* in [1], in which the behavior of a program on a complex CPU is bounded by intermediate deadlines to a known WCET, based on a simple in-order CPU with a cache. Rochange and Sainrat [24] limit the operations of a complex CPU to ensure that basic blocks always have the same execution time irrespective of execution history, thus eliminating timing anomalies. However, their CPU cannot exploit ILP across basic block boundaries, which is known to be a limiting factor [31]. Another approach is to eliminate conditional branches altogether and use predication instead: this “single path” paradigm was proposed by Puschner [21].

Other approaches begin with predictable hardware. Scratchpads [20, 29, 33] are an example of the replace-

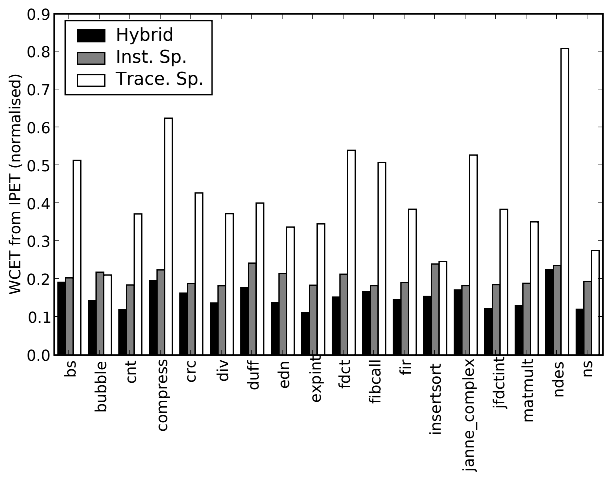


Figure 13. Effects of trace and instruction scratchpads on test program WCETs (calculated using IPET and normalized).

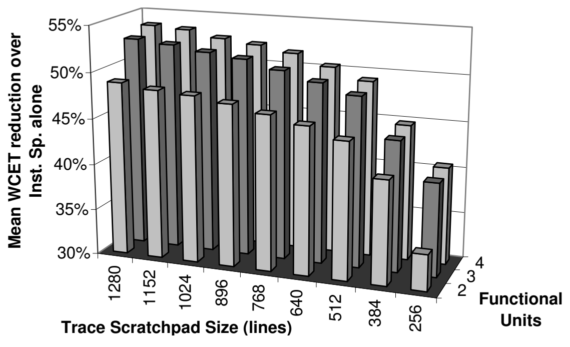


Figure 14. Effects of architectural variations on the WCET reduction achieved by using both types of scratchpad in place of an instruction scratchpad alone. Each data point is the mean reduction across all test programs, and is computed by dividing the instruction scratchpad WCET by the hybrid scratchpad WCET for that program in that architecture.

ment of a dynamic CPU component with a static or explicitly programmed one that is easier to predict. Instruction cache locking [9] uses similar principles, but is easier to implement in some off-the-shelf CPUs which do not provide scratchpad memory. Trace scratchpads extend these principles to control other CPU components, such as additional functional units. The advantage of these approaches is that IPET-based methods may be used to obtain tight estimates of the WCET because program execution times can be statically predicted.

For some programs, it is possible to dispense with a CPU entirely. In [32], Ward and Audsley propose migration of Ada software tasks into FPGA hardware to provide a high speed implementation with predictable timing. However, this cannot scale to programs of any size. One possible solution is hardware virtualisation, as described in [30] by Ullmann *et al.*, where run-time FPGA reconfiguration is used to load tasks, but it is not yet clear how well this will scale to large software-based real-time systems.

8 Conclusion

Our results confirm that instruction scratchpads can provide substantial WCET reductions without introducing analysis difficulties. We have shown that this also applies to trace scratchpads, which can be used to reduce WCETs even further by optimizing WC paths using traces. We have also described an IPET-based algorithm to reduce the WCET of a program, and obtained measurements from a fully working implementation in various architectural configurations. Trace scratchpads are an architectural solution for WCET reduction, forming traces to speed up WC paths as a post compilation step (as opposed to a process applied during compilation, e.g. [38]). They are thus able to address the two issues of CPU timing predictability and program WCET reduction at the same time.

Our results could be improved by optimization of the implementations used. Currently up to 32 extra clock cycles are needed for each trace execution, depending on the number of registers updated, and whether the trace returns to machine code or restarts itself after execution. It is known that instruction scratchpad performance can surpass that of an instruction cache [14], and so it is possible that the most efficient trace scratchpad implementations could surpass the performance of a trace cache (as featured in the Pentium 4), but with fully predictable operation and reduced hardware complexity. Trace caches are dynamically updated by the CPU: trace scratchpads shift this feature into software. Trace scratchpads are an invasive technology that cannot be easily retrofitted onto an existing CPU, but as MCGREP demonstrates [35], FPGA-based soft core CPUs can make use of the feature and can be compatible with existing ISAs.

This paper has not considered updating scratchpad contents at runtime, but previous work [14, 27] indicates that this can be beneficial for instruction scratchpads. Therefore, it is reasonable to expect it will also be beneficial for trace scratchpads, and could allow this approach to WCET reduction to be applied to programs of any size.

9 Acknowledgments

Thanks to Adam Betts for his advice regarding IPET, and to all the reviewers for their helpful comments.

References

[1] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA). In

Proc. RTSS, pages 114–125, Washington, DC, USA, 2004. IEEE Computer Society.

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. RTSS*, pages 172–181, 1994.

[3] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Comput.*, 1(2):179–194, 2005.

[4] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[5] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[6] P. P. Chang, N. F. Warter, S. A. Mahlke, W. Y. Chen, and W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Trans. Comput.*, 44(4):481–494, 1995.

[7] D. Lampret. OpenRISC 1000 Architecture Manual (accessed 26 April 07). `http://www.opencores.org/cvsget.cgi/or1k/docs/openrisc_arch.pdf`, 2006.

[8] S. Edwards and E. A. Lee. The Case for the Precision Timed (PRET) Machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, Nov 2006.

[9] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.

[10] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.

[11] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.

[12] J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[14] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. Exploiting statistical information for implementation of instruction scratchpad memory in embedded system. *IEEE Trans. VLSI*, 14(8):816–829, 2006.

[15] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. DAC*, pages 456–461, New York, NY, USA, 1995. ACM Press.

[16] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proc. RTSS*, page 12. IEEE Computer Society, 1999.

[17] Malardalen WCET research group. WCET Benchmarks (accessed 18 October 07). `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2007.

[18] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3):217–247, 2000.

[19] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.

[20] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[21] P. Puschner. Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures. In *Proc. ECRTS*, Technical Report, York YO10 5DD, United Kingdom, Jun. 2002. Department of Computer Science, University of York.

[22] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[23] P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Syst.*, 13(1):67–91, 1997.

[24] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. CF*, pages 307–314, New York, NY, USA, 2005. ACM Press.

[25] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proc. MICRO*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[26] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359.

[27] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. ISSS*, pages 213–218, New York, NY, USA, 2002. ACM Press.

[28] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. DATE*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[29] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[30] M. Ullmann, M. Hubner, B. Grimm, and J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. *IPDPS*, 04:135a, 2004.

[31] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report WRL-93-6, DEC Western Research Laboratory, 1995.

[32] M. Ward and N. Audsley. Hardware compilation of sequential Ada. In *Proc. CASES*, pages 99–107, New York, NY, USA, 2001. ACM Press.

[33] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. DATE*, pages 600–605, Washington, DC, USA, 2005. IEEE Computer Society.

[34] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[35] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.

[36] J. Whitham and N. Audsley. A self-optimising simulator for a coarse-grained reconfigurable array. In *Proc. UK Embedded Forum*, pages 99–109. University of Newcastle, April 2007.

[37] R. Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. *LNCS*, 2937:309–322, 2004.

[38] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Syst.*, 34(2):129–152, 2006.