2016 IEEE 28th International Symposium on Computer Architecture and High Performance Computing

Value Reuse Potential in ARM Architectures

Rodrigo C. Moura*, Giovane O. Torres*, Maurício L. Pilla*,
Laércio L. Pilla§, Amarildo T. da Costa†, Felipe M. G. França‡
*PPGC – UFPEL. 1 Gomes Carneiro St. Pelotas, Brazil
Email: {rcmoura, gdotorres, pilla}@inf.ufpel.edu.br
§Department of Informatics and Statistics – UFSC. Florianópolis, Brazil
Email: laercio.pilla@ufsc.br
†IME. 80 Gen. Tiburcio Sq. Rio de Janeiro, Brazil
Email: amarildo@cos.ufrj.br
‡COPPE – UFRJ. 2030 Horacio Macedo Ave. Rio de Janeiro, Brazil
Email: felipe@cos.ufrj.br

***Abstract*—Code execution in modern superscalar processors is inherently redundant. Many instructions execute repeatedly with the same inputs, producing the same outputs, thus wasting resources in the process. Value reuse techniques memorize previous executions of instructions, blocks or traces which may be reused if they appear again with the same input contexts. Although trace reuse techniques show great potential for both performance and energy consumption improvement, they have not been studied yet in one of the most widely available computer architectures – the ARM architecture. In this paper, the main issues with reusing traces in instruction sets with conditional execution are revisited. Afterwards, the reuse potential in the benchmark suite MiBench is analyzed varying (i) how traces are generated, and (ii) the size of reuse tables. Our results show that a memoization table of 32 KiB allows to reuse 18.36% of the total instructions on average.**

I. Introduction

Every day, more complex applications are developed, demanding more powerful processors, causing computer designs to increase their complexity. Parallel to this, most of the systems are used for more than a single purpose, thus making development of dedicated hardware not feasible in most cases.

In this context, there is a potential for value reuse techniques. These techniques aim to increase performance of general purpose processors by reducing the number of executed instructions. Reuse techniques make use of value locality, which can be defined as the potential of predicting values of program execution at runtime [10]. Thus, predictability can be defined as:

- Temporal, when it relates to the probability of an execution to generate the same result as another one executed in a near time;
- Spatial, which corresponds to the probability of an execution to generate a variation from an earlier one.

Value locality can be explored during run time to avoid running again redundant tasks, exploiting the same spatial and temporal locality phenomena that allow caches to reduce average latency in memory hierarchies.

According to Gabbay [10], a large portion of many applications is composed by redundant and predictable computation. The exploration of reuse techniques on simulated MIPS processors reached speedups up to 1.28 [29]. Moreover, Tsai and Chen [41] exploited value locality as a way of reducing energy consumption on ARM processors. In this case, the execution redundancy is used in order to increase the number of instructions issued to the processor. Hence, it causes a smaller number of pipeline stalls and it also increases the IPC. They also reported 75% less energy consumption.

In order to allow reuse, an additional buffer is necessary near the processor. This buffer is used to store information about previous executions in order to compare them with future executions. Reuse also requires mechanisms to test the contexts and effectively reuse the instructions.

Considering the need for additional hardware to implement reuse mechanisms, many factors must be evaluated to ensure the viability of the technique. These factors relate primarily to the buffer format and policies for data storage. The structure of the additional hardware for a reuse mechanism must consider the characteristics of the architecture used, which include: instruction set, size and number of registers, forms of address and other specific factors to each architecture.

In this paper, we assess the potential of value reuse techniques at trace level on ARM processor, considering the characteristics of the instruction set and the behavior of execution on this architecture. A behavior analysis of the execution of the benchmark suite MiBench on the ARM architecture simulator Sim-Panalyzer was made to enable this work. From this analysis, this paper proposes a trace reuse strategy and its storage structure, which exposes the impact of limitations during the construction of traces and their effects when using buffers of different sizes. Finally, we present the potential of trace reuse techniques using the evaluated strategies.

This paper is organized as follows. Principles of Value Reuse are described in Section II. In Section III, a short overview of the main features of ARM processors and their main challenges for reuse are presented. Modifications to trace reuse in order to address ARM's peculiarities and the development environment are described in Section IV. Section V presents our results. Related work is discussed in

978-1-5090-6108-2/16 $31.00 © 2016 IEEE
DOI 10.1109/SBAC-PAD.2016.30

Section VI. Finally, conclusions and future work are described in Section VII.

II. Value Reuse

Value Reuse is a technique that exploits execution redundancy by reusing recurring sequences of executions [11]. It is originally designed as a non-speculative technique, i.e., it reuses only known values without making predictions. As the execution of the computation happens, the inputs and outputs of instructions are stored in an indexed table for future access. When this computation is executed again, its inputs will be compared with the ones stored in the table. If they match, output values stored in the table are directly copied to the output registers. Thus, some pipeline stages are not used, which saves resources.

Figure 1 depicts how an instruction reuse mechanism may be attached to an instruction pipeline. The reuse test is done in parallel with instruction fetch so as to avoid slowing the original pipeline stages. Therefore, the addition of a reuse mechanism does not reduce the performance of the architecture, even if instructions are not reused.

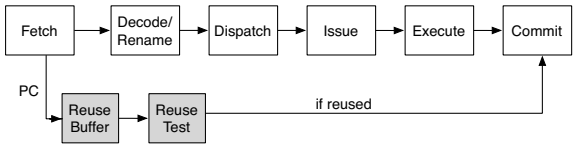

Fig. 1. Pipeline with reuse mechanism [29].

For every instruction in the reuse domain that is decoded, its inputs are compared with candidate instructions from a reuse table. If the inputs of an entry match the current values, its output may be written to the destination registers, thus collapsing dependencies and avoiding redundant execution. If there is no match, the instruction is not reused and continues its regular way in the pipeline.

In addition to the advantages related to the reduction of instructions executed, energy savings, and reduced runtime, it is possible to highlight [6], [39]:

- Results of executions are available earlier, which can anticipate the execution of dependent instructions;
- Data dependencies are reduced because dependent instructions can be reused in parallel; and
- Executions generated by a branch misprediction may be reused by future executions, instead of being discarded.

Throughout the approaches employing value reuse techniques, the basic difference among them is the unit of reuse [16], which can be such as reuse of instructions, basic blocks, or traces. All these strategies try to minimize the costs caused by the additional hardware and to maximize the possibility of reuse.

Traces are dynamic sequences of executed instructions that may contain branches [7], [12], as shown in Figure 2. Regarding reuse, traces have an input and an output context. The input context holds values required by instructions, while the output context stores the set of values generated by the execution of these instructions. To allow reuse, these contexts must be stored in buffers, or memoization tables, in order to represent the traces with potential to be reused.

Just as instruction reuse, traces must pass by a reuse test to determine whether reuse will be made or not. As the instructions are fetched, they are compared with instructions that form traces already stored in reuse buffer. Traces are considered redundant when their input contexts are equal. The approach of trace reuse allows several instructions to be reused at the same time, suppressing redundant sequences of executions.

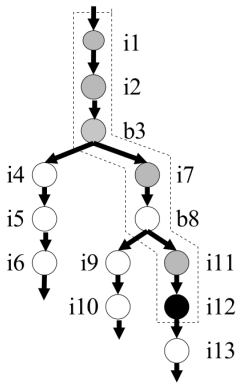

Fig. 2. Trace construction [29]

III. The ARM Architecture

The ARM architecture [37] is a category of superscalar RISC (Reduced Instruction Set Computer) processors that aim for simplicity and low power consumption. These characteristics made these processors the market's reference for embedded computing and handheld devices. They are easily found on mobile phones, smartphones, tablets, calculators, onboard computers, printers, network equipment, etc. As a RISC architecture, it has the following characteristics:

- Large and uniform register file;
- Load/store architecture, where operations occur only between registers, not directly in memory;
- Simple addressing modes, with all load and store addresses determined by contents of registers and instruction parameters; and
- Fixed and uniform instruction size.

One of the main challenges in implementing reuse for ARM architectures is the feature of conditional execution of instructions. In the ARM instruction set [19], instructions can be conditionally executed (predicated). The execution is dependent on the state of conditional bits at run time. Inside the instructions, there are 4 bits that define 16 different conditions for an instruction to be executed, as shown in Table I. An instruction will be executed or not according to the state of these conditional bits of the CPSR (Current Program Status Register). If the corresponding bits on CPSR indicate that the

condition is true, the instruction executes normally, otherwise, it does nothing. The update of conditional bits on CPSR can be done by executing comparison instructions (such as *CMP*), and instructions with the *S* suffix force CPSR bits to update. So, in order to implement trace reuse techniques on ARM architectures, conditional instructions must be considered.

TABLE I
CONDITIONAL SUFFIX ON ARM ARCHITECTURE

Suffix	Description
CS	Carry set
CC	Carry clear
EQ	Equal
VS	Overflow set
VC	Overflow clear
GT	Greater than
LT	Less than
GE	Greater than or equal
LE	Less than or equal
PL	Plus (Positive)
MI	Minus (Negative)
HI	Higher than
LO	Lower than
HS	Higher or same
LS	Lower or same
AL	Always

IV. REUSING TRACES IN ARM ARCHITECTURES

In order to analyze the behavior of programs running on the ARM architecture, experiments were performed with the benchmark suite MiBench on the Sim-Panalyzer simulator. Both are described in Section IV-A, while Section IV-B details the trace reuse mechanism proposed for the experiments.

A. Development Environment

The Sim-Panalyzer simulator [25] is a software used to simulate ARM architectures on general purpose computers. This tool was based on another simulator, named SimpleScalar [4], that is designed to simulate architectures such as x86, PISA, ARM and Alpha. The main objective of Sim-Panalyzer is to create a power estimator in order to evaluate power/performance tradeoffs. The simulator implements the ARMv7 instruction set [1] and the pipeline of a StrongArm processor [46]. Sim-Panalyzer's source code is also available, which was an important requirement for this work. More specifically, we modified the simulator in order to intercept executed instructions as well as register values at each execution. Such modifications do not affect the execution of a program.

The simulated workload is composed by MiBench [14], that is a free and commercially representative set of benchmarks. From this set, 23 applications have been used, which are divided into six categories:

- Automotive and Industrial Control;
- Network;
- Security;
- Consumer Devices;
- Office Automation; and
- Telecommunications.

MiBench also has available pre-compiled binaries for ARM architectures, which streamlined the tests presented in this paper.

B. Trace Reuse Mechanism

For the trace reuse mechanism to be used in the ARM architecture, we first defined the reuse domain, which is the instruction set that is allowed to be reused. Previous studies [9] show that integer instructions without side effects represent on average 66% of all instructions executed by the Sim-Panalyzer simulator using benchmarks from MiBench. Other studies about reuse on MIPS processor observed that including memory instructions in traces requires more complex storage structures and extra mechanisms to control memory data consistency [21]. Furthermore, it was observed that floating-point instructions show less redundancy than integers instructions.

According to these analysis, this study considers only integer instructions without memory accesses as the reuse domain, as they cover the largest instruction group with greater chances of presenting redundancy and that require the smallest and simplest storage structures. This means that only instructions of this group are able to compose traces and, therefore, only those may be reused. This group allows arithmetic and logic instructions, data movement instructions, comparisons, and branches. With this reuse domain, we have determined the basic structure to represent traces. This structure is detailed in Table II.

TABLE II
TRACE ENTRY

Entry	**Description**	Size in bits
pc	address of first instruction in trace	32
npc	next instruction address after trace	32
irb	bitmap of input registers	16
orb	bitmap of output registers	16
icv	input context values	$32\times m$
ocv	output context values	$32\times n$
icpsrb	bitmap of input condition flags	4
ocpsrb	bitmap of output condition flags	4
ipcsrv	values of input condition flags	4
opcsrv	values of output condition flags	4

This structure represents a trace stored in the memoization table. Both m and n represent how many input and output registers are allowed in the trace, respectively. This trace must be composed of a sequence of instructions from reuse domain and it is finished by a instruction outside this domain. In order to represent the beginning and the end of the instruction sequence, the addresses of the first instruction and the instruction immediately after the end of the sequence are stored (pc and npc).

For each instruction in the range from the first to the last instruction inside the trace, the values of registers operated by the instructions and the values generated by their executions are stored. Thus, we have a set of input values, which represents the values operated at the trace (irb and icv, defining an input context) – and a set of output values representing the

values generated by the execution of the trace (orb and ocv, defining an output context).

If there are instructions in the trace that have their execution depending on conditional flags, the state of these flags must also be stored as part of the input context ($icpsrb$ and $ipcsrv$). Furthermore, if any instruction in the trace updates a conditional flag, all modified flags must be stored as part of output context ($ocpsrb$ and $opcsrv$).

Given the aforementioned components of traces, Algorithm 1 details how traces are formed during run time. It is important to emphasize that other restrictions, such as maximum input or output context, may also be established in order to reflect hardware constraints.

The structure shown in Table II and the mechanism for trace formation presented in Algorithm 1 were implemented inside the Sim-Panalyzer simulator. The instructions are intercepted with their operand values and results of executions. With the use of this trace reuse mechanism, it was possible to analyze the behavior of application executions and implement the reuse tests presented in this paper. It is important to note that actual reuse is not performed in the simulator, as the objective of this work is to evaluate the potential for trace reuse on ARM.

Data: Instruction stream I, reuse domain RD, register bank R, condition register $CPSR$.
Result: Trace for reuse or nothing.

$pc \leftarrow$ current PC
$npc \leftarrow$ next PC
$in, out, condr, condw, irb, orb \leftarrow \emptyset$
$n \leftarrow 0$
foreach $insn \in I$ **do**
  if $insn \in RD$ **then**
    $n \leftarrow n + 1$
    $in \leftarrow$ input registers
    $out \leftarrow$ output registers
    $condr \leftarrow$ conditional flags read
    $condw \leftarrow$ conditional flags written
    $npc \leftarrow$ next PC
    foreach $i \in in$ **do**
      if $i \notin irb \wedge i \notin orb$ **then**
        $irb \cup \{i\}$
        $irv[i] \leftarrow \{R[i]\}$
      end
    end
    foreach $o \in out$ **do**
      $orb \cup \{o\}$
      $orv[o] \leftarrow \{R[o]\}$
    end
    foreach $c \in condr$ **do**
      if $c \notin ipcsrb$ **then**
        $ipcsrb \cup \{c\}$
        $ipcsrv[c] \leftarrow CPSR[c]$
      end
    end
    foreach $c \in condw$ **do**
      if $c \notin ipcsrb$ **then**
        $opcsrb \cup \{c\}$
        $opcsrv[c] \leftarrow CPSR[c]$
      end
    end
  else
    if $n > 0$ **then**
      return $(pc, npc, icr, icv, ocr, ocv, ipcsrb, ipcsrv, opcsrb, opcsrv)$
    else
      return $(\emptyset)$
    end
  end
end

Algorithm 1: Trace creation procedure.

V. Trace Reuse Evaluation

This section presents experimental results based on the trace reuse environment and simulator described in Section IV. As reuse mechanisms require extra memory structures close to the processor, similar to L1 caches, the size of the simulated reuse buffers must be limited as it would be the case real hardware implementations. For our experiments, we used FIFO (first in, first out) as the trace replacement policy and limited buffers from 1 KiB to 1024 KiB, in order to cover the most frequently used L1 caches, as well as bigger and smaller sizes for comparison.

With limited resources, efficient ways to store data for reuse must be established. This paper presents the combination of two limitations to the use of memoization tables: *(i)* the maximum number of input and output registers of each trace; and *(ii)* the maximum number of traces inside the memoization table. Considering a fixed buffer size, when changing the maximum number of input and output registers, the size of each trace also changes. This implies a variation in the total number of traces that fit in the buffer. This change in the number of registers causes the following behavior:

- Less registers for trace context:
  - Shorter traces can be formed, decreasing reuse; and
  - More traces can fit in the buffer, increasing reuse.
- More registers for trace context:
  - Longer traces can be formed, increasing reuse; and
  - Less traces can fit in the buffer, decreasing reuse.

With the execution of MiBench benchmarks on the Sim-Panalyzer modified simulator, we have quantified how many traces would be covered for all limitations on input and output registers. Table III presents the percentage of possible traces that can be formed by limiting the number of input or output registers. We can observe that, in order to represent 100% of the possible traces, 11 input registers and 11 output registers are required. We can also see that by using 7 registers for the input context and 7 for output covers almost all traces (≈99%). This reduction would cause small losses in reuse by traces that would not be formed. However, with less registers per trace, the structure for storing each trace becomes smaller,

which allows more traces to fit in the buffer, hence increasing the probability of reuse. Besides, storing less registers in the input context tends to reduce the pressure in the register file during reuse tests.

TABLE III
TRACE DISTRIBUTION BY INPUT/OUTPUT CONTEXT

Registers	Traces – input	Traces – output
1	36.06%	70.82%
2	70.70%	84.47%
3	84.20%	90.78%
4	92.94%	94.77%
5	96.99%	96.52%
6	98.44%	98.07%
7	99.39%	99.08%
8	99.44%	99.23%
9	99.97%	99.90%
10	99.97%	99.96%
11	100.00%	100.00%
12	100.00%	100.00%
13	100.00%	100.00%
14	100.00%	100.00%
15	100.00%	100.00%
16	100.00%	100.00%

Based on this analysis, we searched for the best combination of number of inputs, outputs and number of entries in the reuse table. This required examining, for each buffer size used, all combinations of number of input and output registers and how many traces each combination would cover. The best combination for each buffer size is determined by the highest reuse reached, represented by the percentage of instructions that would not be re-executed.

Table IV presents the combinations that have achieved the best results for each reuse buffer size.

TABLE IV
BEST CONFIGURATIONS FOR EACH BUFFER SIZE

Buffer (KiB)	Trace size (bytes)	Traces	Input Regs	Output Regs	% of traces
1	46	22	4	4	91.50%
2	46	44	4	4	91.50%
4	46	89	4	4	91.50%
8	46	178	4	4	91.50%
16	46	356	4	4	91.50%
32	50	655	5	4	94.53%
64	54	1213	5	5	95.93%
128	82	1598	7	10	99.34%
256	82	3196	7	10	99.34%
512	82	6393	7	10	99.34%
1024	82	12787	7	10	99.34%

According to the configurations shown in Table IV, the best reuse for smaller tables is reached by applying restrictions in the size of input and output contexts, as can be seen on buffers from 1 KiB to 16 KiB. In these cases, the major limitation on reuse is attributed to the lower number of traces that fit in the buffer. However, for larger tablers, the limitation caused by the number of traces in the buffer is less significant. In this way, larger traces, with more input and output registers, can be constructed using buffers from 128 KiB to 1024 KiB.

The aforementioned behavior can be seen in Figure 3, which shows the density of traces by KiB of memory. For the bigger buffers, from 128 KiB to 1024 KiB, the density is 12.5 traces per KiB, while for the smaller buffers it is 22 traces per KiB. This behavior shows that in smaller buffers, the high density of traces aims to compensate the reduced storage space, which is the main factor that limits reuse. This behavior does not appear in buffers larger than 64 KiB. In tables with more space, the low density of traces does not reduce reuse, allowing the use of more registers in the trace structure.

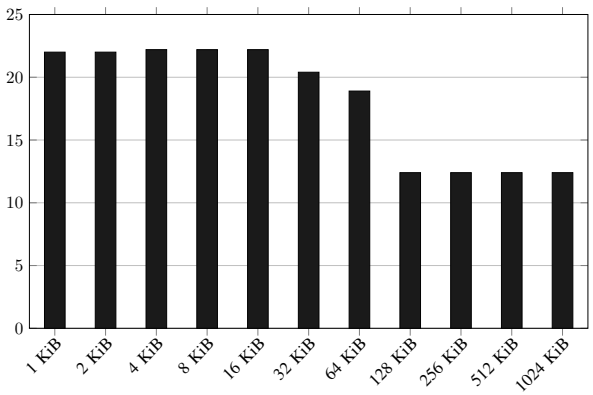

Fig. 3. Density of traces

Figure 4 presents the percentage of reuse achieved by the best settings, representing the portion of redundant computing suppressed by the reuse mechanism. The linearity of reuse with different buffer sizes, even those using different configurations, shows that the strategy for limiting input and output registers makes the use of buffers more efficient. Furthermore, these results confirm the applicability of trace reuse on ARM processors using buffers with similar size to those used in L1 caches.

Without applying limitations on trace formation and storage, it was possible to achieve 32.38% of reuse by designing 400 MiB for reuse tables. On the other hand, a buffer of 32 KiB makes it possible to achieve 18.36% of reuse, which corresponds to more than 56% of the reuse with unlimited tables, but with a fraction of the cost.

VI. RELATED WORK

Many different mechanisms based on value reuse have been proposed. The reuse granularity includes instructions [34], [40], expressions and invariants [24], basic blocks [18], traces [8], [7], [12], as well as instruction blocks and sub-blocks of arbitrary size [17]. The techniques change in terms of their dependence on hardware and compiler support [17], [47].

Pratas et al [30] developed a new constrained loop unrolling technique for reusing instruction inside loops in low power architectures. A controller was designed to identify reusable instructions within loops. This technique is more closely related to Trace Caches [33] than trace reuse, as both do not

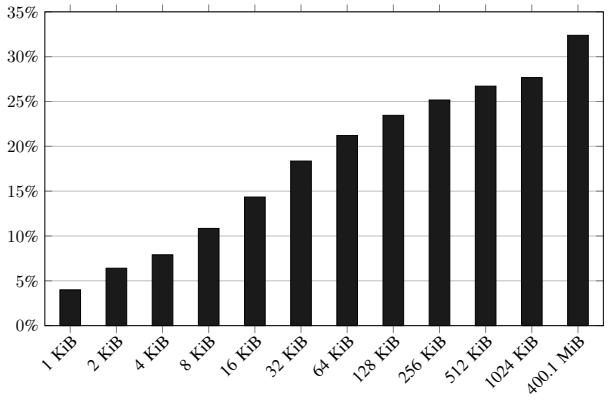

Fig. 4. Reused instructions using buffer limitations

store operands but just the instructions themselves. However, [30] does not require an extra buffer, using the reorder buffer as source of instructions.

Tsumura et al [42] identifies reusable regions by tracking subroutine call and return instructions, or by backward branches in the case of loops. This architecture uses threads running on other cores to validate speculative reuse. Oda et al [26] extended Tsumura's previous work by improving the efficiency of input memoization.

Tsai and Chen [41] proposed a Trace Reuse (TR) Cache to reduce energy consumption in embedded systems. Instead of reusing the computations, TR only reuses the decoded sequences of retired instructions, thus avoiding cache misses and improving overall efficiency. However, this mechanism was evaluated for simple embedded processors and may not scale well when used in more complex processors (such as out-of-order ones).

Rahimi et al [31] uses spatial memoization to reduce the costs of recovering errors in a SIMD architecture. An interesting feature of this work is the absence of reuse tables, where reuse occurs from an instruction running on a strong lane to the ones on the remaining weak lanes. Experimental results showed that 62% of recovery of errant instructions is avoided using their technique. The authors extended their work for floating points in GPUs [32], with results of energy saving from 8 to 28% for error correction.

In most studies, memory accesses are not reused because of side effects and aliasing disambiguation difficulties. One approach to implement load and store reuse is to manage registers as a level in the memory hierarchy [27]. Another approach employs instruction reuse to exploit both same instruction and different instruction redundancy [48].

Load value prediction [23] is a specialization where only loads are predicted, motivated by their long latencies and high value locality. Another version [43] uses the value stored in a register as the prediction, without requiring extra tables.

Many variations on value prediction have been proposed, including two-level value prediction [45], hybrid value prediction [45], [35], and others, many of which were inspired by branch prediction. Value prediction based on correlation [45], [36] uses global information about the path in selecting predictions. Prediction of multiple values for traces [35] may be done for only the live-out values in a trace, reducing necessary bandwidth in the predictor. Speculation control [13] is used to balance the benefits of speculation against other possibilities. Confidence mechanisms [5] are employed to improve value prediction by restricting prediction of unpredictable values.

The upper bound limits of RST architecture (Reuse through Speculation on Traces) in a superscalar architecture with oracle prediction is presented in [28]. Pilla et al. [29] did an extensive study on upper bound limits for feasible architectures, proposing a version with a unified reuse table to further reduce die area requirements. This work was only intended to verify the impact of reducing the reuse domain or by reusing only instructions inside loops in overall performance.

Tune et al [44] suggest finding and predicting chains of dependent instructions in the critical path to optimize performance. Wu et al. [47] proposed speculative reuse but their work requires compiler support.

Huang, Choi and Lilja [15] proposed a scheme that uses a Speculative Reuse Cache (SRC) and Combined Dynamic Prediction (CDP) to exploit value reuse and prediction. During execution, a chooser picks a prediction from the value predictor or a speculatively reusable value from SRC. With this approach, they have achieved a speedup of 10% in a 16-wide, 6-stage superscalar architecture, with 128 KB of storage for the CDP.

Liao and Shieh [22] combined instruction reuse and value prediction, using a reuse buffer and a value prediction table that are accessed in parallel. If inputs are unavailable, the value predictor is used. For a 6-stage pipeline, they achieved an average performance gain of 9%.

The Contrail architecture [20] used two threads, one with speculative execution running on low-latency functional unities, and another thread validating results predicted by the first thread. The speculation thread aims to transform critical paths into non-critical paths, and the execution on slower functional unities allows for reductions of up to 40% on energy dissipation. RST does not require an extra thread to validate results from speculation, making it simpler to implement than other approaches. Trace speculation recovery uses a mechanism similar to those used for branch misprediction.

Auto-memoization for the reuse of loops and functions in ARM processors is proposed by Shibata et al. [38]. They reported that memoization improved performance on an average of 7.9% for a subset of SPEC CPUint 1995. Although they discussed implementation details for their technique in an ARM processor, they did not evaluate the contribution of different instruction types nor they allow for the flexibility of trace reuse.

VII. Conclusions

In this paper we presented a trace reuse strategy for ARM processors using different table sizes. For this purpose, we considered the characteristics of the ARM instructions for

defining memoization table structure; the behavior of the execution of benchmark suite on the ARM simulator Sim-Panalyzer was analyzed in order to define limitations on the construction of traces, aiming to increase the efficiency of buffer use; and finally, the reuse potential was presented with different configurations of traces for different buffer sizes.

Tests showed that the storage structure used to keep track of the traces does not require more than 11 registers for the input context and 11 for the output context. This means that, with such limitation, the maximum possible number of traces can be constructed, corresponding to a structure without restriction. Furthermore, it was observed that even greater restrictions may reach results very close to the maximum possible. This behavior shows that the limitation in the contexts during the construction of traces is a useful strategy to reduce the trace size with little changes in reuse probability.

When implementing different restrictions in input and output contexts of traces, it was concluded that buffers of different sizes achieve their best results with different limitations. Larger buffers achieve better results with fewer limitations in context construction, while smaller buffers are more efficient when applying more restrictions. This behavior is due because of the size of each trace and how many traces each buffer size can cover. Thus, it was concluded that the number of traces at smaller buffers is a key factor for reuse, even if the set of traces with possibility of reuse is smaller.

Finally, the presented results confirm the potential of trace reuse techniques on ARM processors, and demonstrate that it is possible to use realistic buffer sizes, when thinking about hardware implementations. Furthermore, the linearity of reuse results, even in different structures for storing traces, shows the importance of considering the total buffer size when defining the strategy of trace construction and storage.

A. Future work

As a continuation of this work, we intend to evaluate the reuse potential by restricting the use of buffer according to trace length. Since one of the advantages of reusing traces is the possibility to suppress instruction sequences in a single moment, reusing longer traces tends to get better results. However, due to the greater number of inputs and outputs of longer traces, the probability of redundant traces decreases. Thus, it is important to evaluate the impact of traces of different lengths in reuse mechanisms.

Trace replacement policies are an important factor that impacts reuse results. Strategies such as a TTL field (time-to-live) for each trace, LRU policy (Least Recently Used) or a FIFO structure (first in, first out) must be compared. So, in order to determine what is the best policy, it is necessary to evaluate two main redundancy features: *(i)* how many times on average each trace is reused; and *(ii)* how much time there is between the trace storage and its reuse.

Another factor that needs to be evaluated relates to the availability of input context values. When making the reuse test, the values of registers may not yet be available to make the comparison. In this case, reuse would be delayed. Pilla et al. [29] proposed the use of value speculation to provide input values not available yet, which achieved positive results in the tests performed on MIPS architectures, and this strategy may be evaluated on ARM processors.

When adding a memory structure at the same level of an L1 cache, the cost of the processor increases. Thus, it is important to compare the performance gains from the addition of the reuse mechanism memory and the increment of the size of the L1 cache. According to the performance of the two approaches, there is a possibility of using part of the L1 cache to implement a reuse buffer without the addition of any extra memory. Thus, instead of focusing on increasing processing power, it is possible to explore the reduction of the computation to reduce the energetic consumption, keeping the original processing power.

Afterwards, we intend to implement a deeper trace reuse mechanism in the Sim-Panalyzer simulator in order to evaluate both performance and energy consumption of trace reuse for ARM architectures.

ACKNOWLEDGMENT

This work was supported by CAPES/Brasil (Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

REFERENCES

[1] ARM7TDMI - Technical Reference Manual. Available at: ⟨http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf/⟩. Last access: Oct 2015.

[2] *Proc. of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, Dec. 1997. Los Alamitos, IEEE Computer Society.

[3] *Proc. of the 26th Annual International Symposium on Computer Architecture*, Atlanta, May 1999. New York, ACM.

[4] T. Austin. *SimpleScalar LLC*. 1997. Available at: ⟨http://www.simplescalar.com/⟩. Last access: July 2015.

[5] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proc. of the 26th Annual International Symposium on Computer Architecture* [3], pages 64–74.

[6] A. T. D. Costa. *Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador*. Phd thesis, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 2001.

[7] A. T. D. Costa, F. M. G. França, and E. M. C. Filho. The dynamic trace memoization reuse technique. In *9th PACT, IEEE CS*, pages 92–99, 2000.

[8] A. T. da Costa and F. M. G. França. The reuse potencial of trace memoization. Technical Report ES–498/99, COPPE–UFRJ, Rio de Janeiro, May 1999.

[9] G. de Oliveira Torres. Um estudo sobre os efeitos da técnica de reuso de traços em arquiteturas arm nas questões de desempenho, 2015. Monografia (Bacharel em Ciência da Computação), UFPel (Universidade Federal de Pelotas), Pelotas, Brazil.

[10] F. Gabbay. Speculative execution based on value prediction. Technical report, EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[11] A. González, J. Tubella, and C. Molina. The performance potential of data value reuse. Technical report, 1998.

[12] A. Gonzalez, J. Tubella, and C. Molina. Trace-level reuse. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 30–37, 1999.

[13] D. Grunwald, A. Klauser, S. Manner, and A. Plezskun. Confidence estimation for speculation control. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 122–131, Barcelona, June 1998. New York, ACM.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[15] J. Huang, Y. Choi, and D. Lilja. Improving value prediction by exploiting both operand and output value locality. Technical Report Technical Report ARCTic 99-06, Laboratory for Advanced Research in Computing Technology and Compilers, July 1999.

[16] J. Huang and D. J. Lilja. Exploring sub-block value reuse for superscalar processors. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:100, 2000.

[17] J. Huang and D. J. Lilja. Exploring sub-block value reuse for superscalar processors. In *Proc. of the 9th International Conference on Parallel Architectures and Compilation Techniques*, pages 100–110, Philadelphia, Oct. 2000. Los Alamitos, IEEE Computer Society.

[18] J. Huang and D. J. Lilja. Extending value reuse to basic blocks with compiler support. *IEEE Transactions on Computers*, 49(4):331–347, Apr. 2000.

[19] P. Knaggs and S. Welsh. *ARM: Assembly Language Programming*. School of Design, Engineering & Computing, Bournemouth University, 2004.

[20] T. Koushiro, T. Sato, and I. Arita. A trace-level value predictor for contrail processors. *SIGARCH Comput. Archit. News*, 31(3):42–47, 2003.

[21] L. S. Laurino, M. L. Pilla, T. S. G. dos Santos, and P. O. A. Navaux. Reuso de traços com loads em arquiteturas superescalares. *WSCAD*, 2005.

[22] C.-H. Liao and J.-J. Shieh. Exploiting speculative value reuse using value prediction. In *Proc. of the 7th Asia-Pacific Conference on Computer Systems Architecture*, pages 101–108, Melbourne, 2002. Australian Computer Society, Inc.

[23] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, Dec. 1996. Los Alamitos, IEEE Computer Society.

[24] C. Molina, A. González, and J. Tubella. Dynamic removal of redundant computations. In *Proc. of the 13th ACM International Conference on Supercomputing*, pages 474–481, Rhodes, 1999. New York, ACM.

[25] T. Mudge, T. Austin, and D. Grunwald. *The SimpleScalar-Arm Power Modeling Project*. Available at http://web.eecs.umich.edu/$\sim$panalyzer/⟩. Last access: Jan 2014., 2001.

[26] R. Oda, T. Yamada, T. Ikegaya, T. Tsumura, H. Matsuo, and Y. Nakashima. Input entry integration for an auto-memoization processor. In *ICNC*, pages 179–185. IEEE Computer Society, 2011.

[27] S. Önder and R. Gupta. Load and store reuse using register file contents. In *Proc. of the 15th ACM International Conference on Supercomputing*, pages 289–302, Sorrento, Italy, 2001. New York, ACM.

[28] M. L. Pilla, B. R. Childer, A. T. da Costa, F. M. G. França, and P. O. A. Navaux. A speculative trace reuse architecture with reduced hardware requirements. In A. F. de Souza, R. Buyya, and W. M. Jr., editors, *Proc. of the 18th Symposium on Computer Architectures and High Performance Computing*, pages 47–54, Ouro Preto, Oct. 2006. Los Alamitos, IEEE Computer Society.

[29] M. L. Pilla, B. R. Childers, F. M. G. França, A. T. da Costa, and P. O. A. Navaux. Limits for a feasible speculative trace reuse implementation. *IJHPSA*, 1(1):69–76, 2007.

[30] F. Pratas, G. Gaydadjiev, M. Berekovic, L. Sousa, and S. Kaxiras. Low power microarchitecture with instruction reuse. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 149–158, New York, NY, USA, 2008. ACM.

[31] A. Rahimi, L. Benini, and R. Gupta. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 60(12):847–851, Dec 2013.

[32] A. Rahimi, L. Benini, and R. K. Gupta. Temporal Memoization for Energy-efficient Timing Error Recovery in GPGPUs. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 100:1–100:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[33] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[34] A. Roth and G. S. Sohi. Register integration: A simple and efficient implementation of squash re-use. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 223–234, Monterey, 2000. Los Alamitos, IEEE Computer Society.

[35] R. Sathe, K. Wang, and M. Franklin. Techniques for performing highly accurate data value prediction. *Microprocessors and Microsystems*, 22(6):303–313, Nov. 1998.

[36] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. of the 30th Annual International Symposium on Microarchitecture* [2], pages 248–258.

[37] D. Seal. *Arm Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., 2000.

[38] Y. Shibata, T. Tsumura, T. Tsumura, and Y. Nakashima. An implementation of auto-memoization mechanism on arm-based superscalar processor. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–8, Oct 2014.

[39] A. Sodani. *Dynamic Instruction Reuse*. University of Wisconsin–Madison, 2000.

[40] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 205–215. Los Alamitos, IEEE Computer Society, 1998.

[41] Y.-Y. Tsai and C.-H. Chen. Energy-efficient trace reuse cache for embedded processors. *IEEE Trans. VLSI Syst.*, 19(9):1681–1694, 2011.

[42] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima. Design and evaluation of an auto-memoization processor. In H. Burkhart, editor, *Parallel and Distributed Computing and Networks*, pages 230–235. IASTED/ACTA Press, 2007.

[43] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *Proc. of the 26th Annual International Symposium on Computer Architecture* [3], pages 270–281.

[44] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proc. of the 7th International Symposium on High Performance Computer Architectures*, pages 185–195, Monterey, Jan. 2001. Los Alamitos, IEEE Computer Society.

[45] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proc. of the 30th Annual International Symposium on Microarchitecture* [2], pages 281–290.

[46] R. T. Witek and J. Montanaro. Strongarm: A high-performance arm processor. In *COMPCON*, pages 188–191, 1996.

[47] Y. Wu, D.-Y. Chen, and J. Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *Proc. of the 28th Annual International Symposium on Computer Architecture*, pages 98–108, Göteborg, Sweden, June 2001. New York, ACM.

[48] J. Yang and R. Gupta. Load redundancy removal through instruction reuse. In *Proc. of the 29th International Conference on Parallel Processing*, pages 61–68, Toronto, Aug. 2000. Los Alamitos, IEEE Computer Society.