Instruction-Level Loop Perforation

Daniela Catelan[1], **Liana Duenha**[1], **Ricardo Santos**[1], **Lucas Wanner**[2]

[1]Federal University of Mato Grosso do Sul (UFMS)
College of Computing (FACOM)

[2]University of Campinas (UNICAMP)
Institute of Computing

{daniela.catelan, liana.duenha, ricardo.santos}@ufms.br, wanner@unicamp.br

Abstract. *Approximate computing (AC) offers techniques ranging from application to circuit levels. AC techniques offer better performance at the cost of inaccurate results. A widely used software AC technique is loop perforation (LP). This paper presents an Instruction-Level LP (ILLP) approach that relies on approximate hardware instructions. We extended the ACCEPT compiler and SPIKE simulator workflows to generate and simulate applications with ILLP. We evaluated the technique comparing the results of precision, number of instructions, cycles, and energy consumption. ILLP achieves a* 74.61% *reduction in the number of instructions for the PI application, a* 51.40% *reduction in the number of cycles for the FFT, and an energy saving of* 74.49% *for the PI.*

1. Introduction

The end of Dennard scaling and the increasing difficulty of fitting more cores into increasingly limited energy budgets have pushed designers to look for new alternatives to offer performance with energy and cost constraints. One potential solution is to relax the quality of output constraints, by designing systems and hardware where it is acceptable that the outputs will have inaccuracy in some cases. Approximate computing (AC) has stood out among researchers for offering techniques ranging from the application level to the circuit level [Catelan et al. 2022].

AC has become promising for tasks that may have some margin of error in the final result in exchange for better performance. However, the use of approximations is still a matter for fault-tolerant applications, such as multimedia or data mining. Most AC techniques aim to solve specific problems or require excessive intervention from the programmer, who needs to identify which parts of the application are susceptible to approximations [Reis and Wanner 2021].

The loop perforation (LP) technique is widely used in AC. LP is simple and effective in reducing the amount of computational work by skipping loop iterations and exchanging precision for other benefits such as performance and energy consumption. Research works [Reis and Wanner 2021, Sidirogloy-Douskos et al. 2011, Li et al. 2018, Moreno et al. 2021, Rodriguez-Cancio et al. 2019] have proposed different ways to apply LP, from a simple loop step up to the adoption of heuristic approaches to find the best increment value.

One common limitation of LP is that once the perforation degree (pd) is established, the application metrics will improve only at the cost of accuracy. A higher de-

gree will perform better at the cost of accuracy, but any change in the degree of perforation requires recompilation. One way to overcome this limitation would be to adopt a strategy where the pd could use approximate hardware resources. This would bring more flexibility to the technique, allowing greater performance and energy savings. A hardware-supported approach could achieve this flexibility goal without forcing any further compilation step or changes to the application code.

This paper presents an Instruction-Level LP (ILLP) approach that relies on approximate hardware instructions. Instead of just setting a pd in the code to skip loop iterations, our approach skips loop iterations by approximate hardware. The idea is to use new approximate instructions that will be responsible for calculating the next loop iteration. Contrary to software LP, our technique, Instruction-Level Loop Perforation (ILLP), calculates the next loop iteration value using approximate hardware. The direct impact of the technique is on the application performance once the approximate hardware runs faster than the exact hardware.

We extended the ACCEPT [Sampson et al. 2015] compiler, the RISC-V toolchain [RISC-V 2022] and the SPIKE simulator [SPIKE 2019] to support a new set of approximate instructions. We introduce a new coarse adder instruction called **addx**, following the format of the RISC-V instruction set. The instruction was defined with the R type format and implemented using the approximate adder InXA1 [Almurib et al. 2016, Catelan et al. 2022] whose logical expression for the output of $S = A \oplus B \oplus Cin$.

To implement this new feature, we extended the ACCEPT with a new workflow that allows users to annotate the pd, the loop to be perforated, and the ILLP. As a result, the output code from ACCEPT now includes annotations for loops to be perforated according to ILLP. We conducted experiments on 13 general-purpose applications and subjected each to different pd using the original LP software and the ILLP. Our approach demonstrated a reduction in the number of instructions for all applications at all levels of perforation while maintaining the same level of precision.

This paper is organized as follows. Section 2 introduces the LP approximation technique; Section 3 presents and discusses related works that use the AC technique, LP; The instruction-level LP design is shown in Section 4; Section 5 describes the applications and experiments performed to evaluate and validate the approximation techniques; The results and discussion of the AC, ILLP, and AS are presented in Section 6; Finally, the conclusions are in Section 7.

2. Loop Perforation Approximation Technique

AC is a design alternative that offers performance with increasingly stringent energy and cost constraints. The possibility to exchange reduced accuracy in results for gains in performance and energy consumption has increased interest in this field of study [Sidirogloy-Douskos et al. 2011]. AC presents approximation techniques for hardware and software designs. In hardware, techniques range from the circuit level to architecture, offering reductions in area and energy consumption in the integrated circuit by employing approximate logic circuits, voltage scaling, and memory density.In software, it achieves better system performance by performing less computational work and reducing accesses to memory [Catelan et al. 2022].

LP is an AC technique that has been gaining appreciation among designers

because it is simple, general purpose, and widely applicable to different applications [Li et al. 2018]. LP consists of skipping loop iterations to reduce computational workload and gain performance. A simple change in the loop step variable is enough to change its performance, but manually changing one or more loops of an application sometimes becomes more costly than the loop execution itself.

The LP requires a pd parameter that indicates how often to skip an iteration at runtime. Figure 1 shows an original loop(Fig. 1(a)), suitable for perforation, along with a code snippet of the perforated loop(Fig. 1(b)) based on a pd. The larger the value of pd, the fewer iterations the loop will execute, which affects performance and energy consumption.

```
1  for(int i = 0; i < b; i++)
2  {
3    loop_body();
4  }
```

(a) Original loop.

```
1  for(int i = 0; i < b; i+=2^pd)
2  {
3    loop_body();
4  }
```

(b) Loop perforation.

Figure 1. Original loop and after applying loop perforation.

Common challenges in adopting LP rely on discovering which loop to perforate and how much to perforate. These challenges are the motivation for a variety of research work that focuses on automatizing the process of finding loops able to be perforated or even adopting different approaches to set the perforation level.

3. Related Work

ACCEPT automatizes the process to find suitable loops for perforation. ACCEPT allows the automatic or programmatic application of various approximation techniques. The ACCEPT considers a loop with a precise and pure body, without early exits and conditionals, fit for perforation. To perform LP, ACCEPT inserts a counter into its code, dividing the loop into three blocks: the header, the body, and the latch. Loop calculations are performed inside the body, and the header and latch are responsible for changing condition variables and checking jump conditions. The counter is the pd, which is a power of 2 [Reis 2021].

In [Sidirogloy-Douskos et al. 2011], space exploration is performed to find the best match between tunable loops through Pareto optimal policies. Critical loops are filtered out once, so just inserting the perforation could lead to unacceptable computation, crash, increased execution time, or memory error. The exploration of the perforation space searches only the Pareto optimal variants, maximizing performance with a specific precision loss limit. The loop space exploration results present a vast region occupied by the variants generated in the exploration space, allowing a wide range of choices and the ability to increase performance in exchange for small, less than 10%, accuracy losses.

Selective dynamic LP [Li et al. 2018] skips selected statements in dynamically selected loop iterations. It selects instructions and iterations based on the behavior of the program by combining offline and runtime analysis, analyzing the control flow, and creating a profile that will be transformed. The results show that selective dynamic LP achieves speedups from $2.89\times$ up to $4.07\times$ with less than 10% loss of precision.

The fault tolerance technique involves running replicas of a task at different times. The method proposed in [Moreno et al. 2021] seeks to approximate tasks using what the authors call Simplified Iterations. The work implies not omitting any iterations, as happens in the LP, but replacing the calculation of some iterations with less complex operations with results close to the original ones. By injecting faults into a set of 3 applications,the technique achieved up to $5.28\times$ fewer approximation errors compared to the original source code.

While most research works look for the best loops to pierce, Approximate Unrolling [Rodriguez-Cancio et al. 2019] focuses on loops that map a function onto the elements of an array. The optimization consists in adding code that interpolates the results of less costly previous iterations, looking to reduce execution time and energy consumption. The code transformation uses the Nearest-Neighbor and Linear Interpolation strategy. The technique improved runtime and power consumption in x86 code by approximately 50% to 110%, with accuracy within acceptable levels.

Related work on the LP subject presents a wide range of techniques. Perforation strategies range from simply determining the loop iteration value larger than the original loop to using more advanced techniques. All the techniques presented are software-based solutions. By exploring an approximate instruction set, our work brings a proposal to carry out the LP in hardware. To the best of our knowledge, this is the first proposal for the LP technique that takes advantage of the approximate hardware (and instructions) available in the processor. One may observe that our LP approach also constitutes a new usage for approximate instruction sets.

4. Instruction-Level Loop Perforation Design

Once the compiler selects the loop to be punctured, the iteration value for the variable will be equal to 2^{pd}, Figure 1(b). The pd is the exponent for the loop iterator to be 2^{pd}, so that when $pd = 1$, in the application, the iteration of the "real" loop will be 2^1=2. The values of pd adopted in the applications are 1, 2, 4, and 8. Actual perforation values are 2, 4, 16, and 256. It is important to note that code approximation only occurs when pd differs from the initial loop iteration step.

Our proposal presents a new LP in which the loop iteration value is not fixed but determined by an approximate instruction. Figure 2(a) provides an example of the ILLP technique being inserted into a loop with an incremental loop step. The original operation for the loop step is replaced with a function that invokes an approximate instruction. The approximate function ADDX performs the approximate adder instruction (**addx**).

Figure 2(b) presents the assembly code (command `volatile asm`) for the ADDX function. Lines 4-8 indicate that an instruction called **addx** will be executed where parameters i and pd will be placed in registers x and y, respectively. The result of the instruction (register z) will be available in the ADDX variable. An offset code (lines 10-11) is used to correct the statement result when it is less than or equal to the current loop step. Note that this offset code is applied for loops with increasing iterator. For a descending loop step, line 10 should evaluate whether the result is greater than or equal to i, and line 11 must be changed to a subtraction operation.

The correction_factor (CF), lines 10-11, could be performed directly in the hardware, but the designer should be aware that CF is only valid to ensure that increment (i) is

```
for(int i = 0; i < b; i=ADDX(i,2^pd))
{
  loop_body();
}
```

(a) Loop perforation with ADDX.

```
int ADDX (int i, int 2^pd)
{
    int ADDX;
    asm volatile {
      "addx %[z], %[x]; %[y]\n\t"
      :[z] "=r" (ADDX)
      :[x] "r" (i), [y] "r" (2^pd) );
    }
    // correction_factor
    if (ADDX <= i)
       ADDX = i + 2^pd;
    return (ADDX);
}
```

(b) ADDX approximate function.

Figure 2. LP with ADDX function call and ADDX function.

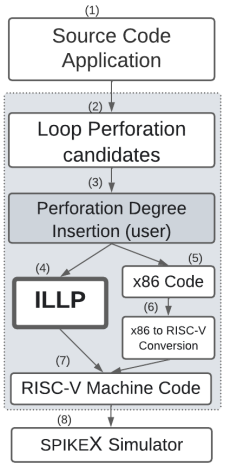


Figure 3. Workflow and toolset to apply the ILLP technique.

always progressing. A CF in hardware will be unnecessary for applications that may use the approximate instruction outside the context of LP.

We extended the ACCEPT to allow users to choose the ILLP in a loop. The first step consists of choosing which loop to perforate. Figure 3 illustrates the steps to use ILLP. Given a source code, ACCEPT will analyze the code and identify all loops that can be perforated. A text file is generated presenting the candidate loops to the user (2). In step (3), the user chooses the loop, the pd, and the LP: ACCEPT_SPIKE (AS) or Instruction-Level Loop Perforation (ILLP).

At this point, the compilation flow is divided into two paths: step (4) means that the ILLP now executes the LP, and the output is RISC-V assembly code; step (5) is the standard LP available in ACCEPT. Step (6) is carried out when the AS is chosen. The conversion is required once that ACCEPT is integrated into LLVM version 3.2 which only generates x86 code. Step (7) converts the assembly to machine code. In step (8), the application can be simulated in the SPIKEX simulator (our extension for the SPIKE simulator). SPIKEX supports approximate instructions, such as **addx**, and is fully compatible with the original SPIKE.

When analyzing a RISC-V assembly code snippet with LP, with $pd = 2$ and a loop limit of 100, for both techniques, we verify that the conversion step of the AS workflow does not introduce additional instructions to the code assembly. However, we noticed that

Table 1. Applications summary. Input apps, Output apps, LP in Hotspots, Has nested loops, Position LP).

APPS	Input	Output	LP in Hotspots	Nested loop	Position LP
BINARY_SEARCH	✓	✓			
CONV1D	✓	✓	✓	✓	O
CONV2D	✓	✓	✓	✓	O
DIJKSTRA	✓	✓	✓		
FANNKUCH					
FFT	✓	✓		✓	O
FIBONACCI			✓		
FLOYD	✓	✓	✓	✓	I
MEDIAN	✓	✓	✓		
MULT100X100	✓	✓	✓	✓	O
NBODY			✓	✓	O
PI			✓		
SPECTRALNORM				✓	I

using ACCEPT can increase the number of instructions to ensure that puncture does not affect the loop body. In this example, ILLP code executes 28.7% fewer instructions (693) than AS (973).

5. Experimental Setup

We have designed and conducted experiments to evaluate and validate the ILLP by comparing its accuracy, number of instructions, cycles, and energy (μWsec) to the LP in the ACCEPT (AS) and the original code (baseline - BL). The experiments were performed across a set of 13 applications[1]: BINARY_SEARCH (BS) [Silveira et al. 2022], CONV1D [Silveira et al. 2022], CONV2D [Silveira et al. 2022], DIJKSTRA (DIJ) [GeeksforGeeks 2022], FANNKUCH-REDUX (FAN) [Game 2022], FFT [Santos 2022], FIBONACCI (FIB) [Silveira et al. 2022], FLOYD-WARSHALL (FLOYD) [Boyini 2022], MEDIAN [Silveira et al. 2022], MULT100X100 (MULT), NBODY [Game 2022], PI [Statescu 2022], and SPECTRALNORM (SPECTRAL) [Game 2022].

We have identified the hotspot function in each application using GProf [Graham et al. 1982]. For applications such as BS, FAN, FFT, and SPECTRAL, which did not have a loop in their hotspot functions, we added perforation to the loop where the hotspot function is called to enhance the use of LP. In cases where the function had a nested loop, we selected the loop with the least impact on accuracy. Table 1 provides a summary of each application, including information on the presence of input and output vectors or matrices, the existence of LP in the hotspots function, whether there is a nested loop, and where the LP is placed (innermost (I) or outermost (O)).

Our experiments used the infrastructure provided by the ACCEPT to insert LP in the code. We simulated all the applications in the SPIKEX RISC-V simulator, following the workflow illustrated in Figure 3. We evaluated the perforation on four degrees ($pd = 1$, 2, 4, and 8). Our comparison involved the ILLP results with the ACCEPT LP and the original (BL) code. Importantly, we inserted perforations on the same loops of each application for both the ACCEPT and the ILLP approaches.

We gathered all the necessary information about the application's performance

[1]Applications are available for download at: https://github.com/lscad-facom-ufms/ILLP-Loop_Perforation

Table 2. Output, Number of Instructions, Number of Cycles, and Energy.

Applications	Output	Instructions	Cycles	Energy (μWsec)
BINARY_SEARCH	500	2,259,486	2,993,337	56.89
CONV1D	1,002	5,144,208	7,180,610	155.20
CONV2D	100	207,223	277,491	5.67
DIJKSTRA	100	1,087,879	1,380,358	27.86
FANNKUCH	38	1,072,887,156	1,274,710,666	25,210.72
FFT	1,024	541,575,553	700,790,974	14,111.47
FIBONACCI	102,334,155	82,678	106,175	2.10
FLOYD	324	253,433	316,767	6.48
MEDIAN	1,000	2,232,393	2,879,240	53.48
MULT100X100	10,000	7,496,393	8,599,491	185.89
NBODY	-0.1690876	318,869,815	422,378,381	8,565.30
PI	3.1480	24,378,464	32,613,086	654.97
SPECTRALNORM	1.2741930	331,087,512	447,130,079	8,932.78

using Prof5 [Silveira et al. 2022], a RISC-V profiling tool that uses the SiFive E24 RV32IMAFBC microcontroller. The SiFive E24 is a high-performance RISC-V microcontroller design model with a 3-stage pipeline running at 125MHz and served to perform power modeling of some instructions. We emphasize here that the simulation of the applications was carried out by SPIKE, which follows the design of a 5-stage RISC-V. Prof5 uses the executable file generated by the SPIKE. Prof5 allowed us to create detailed profiles of RISC-V programs from the SPIKEX log. The profile data generated by Prof5 is the number of cycles, instructions, power, energy, and average power per cycle. We also customize the energy model by entering new instructions and creating custom ones. Approximate instructions like **addx** have also been added to the power model, with an energy savings of 13.92% [Catelan et al. 2022] compared to the default add custom instruction by Prof5.

Table 2 presents the results of output, number of instructions, number of cycles, and energy (μWsec) of the original applications (BL). It may be observed that the FAN application has the highest number of instructions among all applications $(1,072,887,156)$ and the highest number of cycles. The FAN, FFT, NBODY, and SPECTRAL applications have the highest energy values.

Column Output represents the original (Baseline - BL) output of each application. BS, CONV1D, CONV2D, DIJ, FFT, FLOYD, MEDIAN, and MULT have matrices or arrays elements as outcomes. FAN, FIB, NBODY, PI, and SPECTRAL have a single number of the output result. When using the LP, we performed the comparison element by element. For example, the MULT application with LP and $pd = 1$ has an output of $5,000$ elements, an accuracy loss of 50%.

The precision metric is the first to consider when comparing approximate optimization techniques. We calculated the relative error ($RE = \frac{|AO-BO|}{|BO|}$) of the LP and comparing them to the output results of the baseline (BO) and the Approximate Output (AO). A larger RE indicates greater imprecision.

6. Results and Discussion

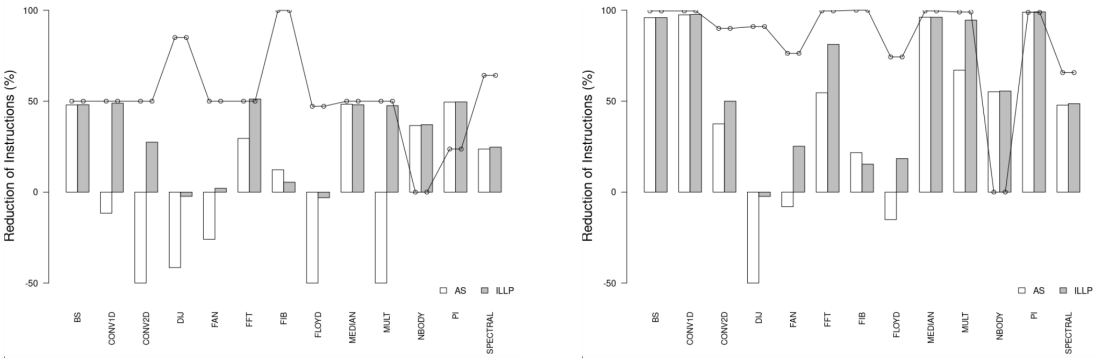
Table 3 presents the RE of the applications in each of the techniques. Note that with $pd = 1$, the BS application presented an RE of 0.5000, which is consistent with the

Table 3. Relative Error.

Applications	pd=1	pd=2	pd=4	pd=8
BINARY_SEARCH	0.5000	0.7500	0.9360	0.9960
CONV1D	0.5000	0.7495	0.9371	0.9960
CONV2D	0.5000	0.7000	0.9000	0.9000
DIJKSTRA	0.8500	0.8600	0.8600	0.9100
FANNKUCH	0.5000	0.7368	0.7632	0.7632
FFT	0.5000	0.7500	0.9375	0.9961
FIBONACCI	0.9999	1.0000	1.0000	1.0000
FLOYD	0.4722	0.6420	0.7253	0.7438
MEDIAN	0.5000	0.7500	0.9360	0.9960
MULT100X100	0.5000	0.7500	0.9300	0.9900
NBODY	0.0002	0.0002	0.0002	0.0002
PI	0.2379	0.3784	0.8420	0.9885
SPECTRALNORM	0.6424	0.6557	0.6579	0.6579

drilling that was drilled at 50% of its BL. As the value of pd increases, the RE also increases, causing a significant precision loss, as in CONV1D, which presents an RE of 0.9960 with $pd = 8$.

Figure 4 presents the percentage reduction of AS and ILLP instructions compared to BL with $pd = 1$ and $pd = 8$. The reference (zero) represents the instructions of the BL application. Upside-down columns indicate that the number of run instructions of the LP is greater than the BL application. The circled line shows the error percentage of each application and technique, allowing us to evaluate the instruction reduction and the impact on the RE. There is a decrease in the number of instructions in most applications as pd increases, just as expected. There is a decrease in the number of instructions in 11 out of 13 applications (highlighted) by the ILLP compared to the BL.



(a) $pd = 1$. (b) $pd = 8$.

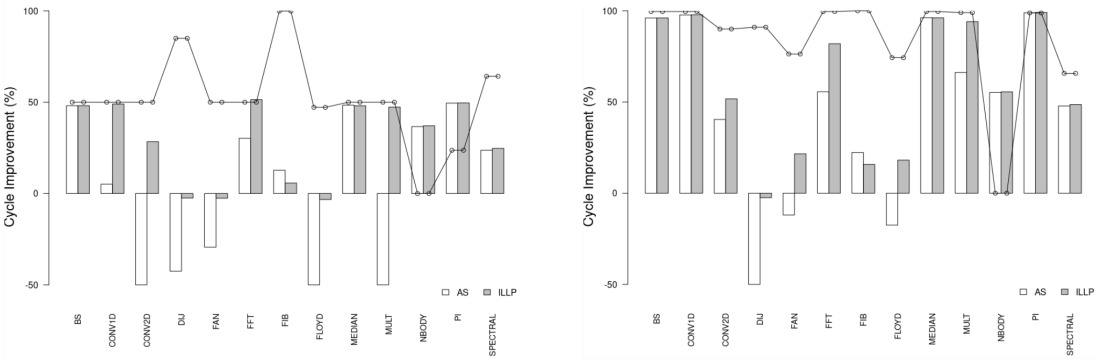
Figure 4. AS and ILLP instructions percentage reduction for each application.

Application DIJ had no significant reduction in instructions over the pd. This application has loops with a small number of iterations so that the pd level does not impact the number of generated instructions. The ILLP has shown increased instructions

compared to BL for DIJ and FLOYD applications. Specifically, the DIJ application had an increase of 2.30% in ILLP_1, while AS_1 had an increase of 41.48%.

The AS presents an instruction increase in 6 applications compared to BL. The MULT application has an unexpectedly large number of instructions. One explanation for this behavior could be the ACCEPT workflow that makes excessive calls to the internal functions $\_mulsi3$ (5016 times) and $\_muldi3$ ($1,000,280$ times), while the ILLP calls the $\_mulsi3$ function 16 times and the function $\_muldi3$ is not used. Application CONV2D shows an increase of 98.66%. In contrast, CONV2D with the ILLP reduced 27.46%. The FFT application presents a decrease of 51.19% (ILLP_1) compared to the BL code. Meanwhile, the AS technique reduced 29.55% (AS_1) compared to BL.

Figure 4(b) shows a significant reduction in instructions with $pd = 8$ in most applications. The FAN application achieves a reduction of 25.22% with ILLP but an increase of 8.08% with AS. On the other hand, the FLOYD application shows an increase in the number of instructions with AS (for all pd) but an instruction decrease using ILLP with $pd = 2, 4, 8$.



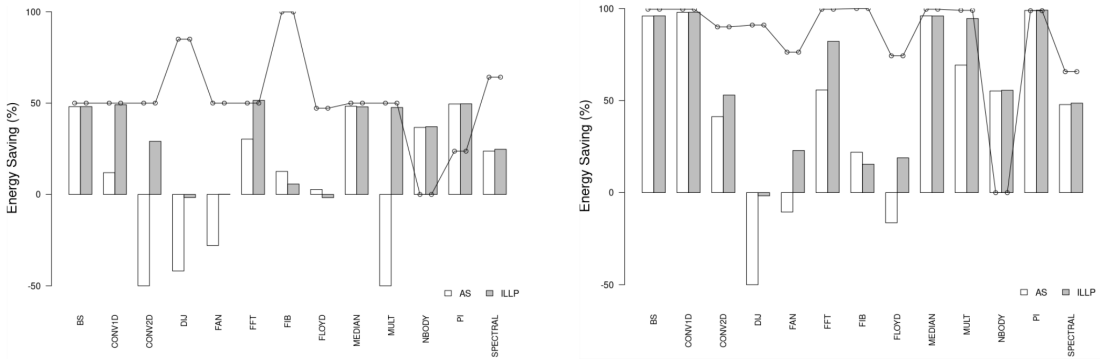
(a) $pd = 1$. (b) $pd = 8$.

Figure 5. AS and ILLP cycle improvement percentage for each application.

Figure 5 presents the cycle improvement percentage for each application using $pd = 1$ and $pd = 8$. The reference (zero) represents the cycles of the BL application. Figure 5(a) shows both AS and ILLP able to achieve a similar cycle reduction for the MEDIAN application, with values of 48.39% and 48.06%, respectively. CONV2D, DIJ, FAN, FLOYD, and MULT applications showed an increase in the number of instructions, leading to an increase in the number of cycles when running on AS. On the other hand, most of the applications that used ILLP showed cycle improvements. The FFT application, in particular, achieved the most significant cycle reduction (51.40%).

The cycle improvement percentage with $pd = 8$ (Figure 5(b)) is quite impressive. Notably, cycle reduction gains are greater than 50% and even reach 99.10% for the PI application. The BS, CONV1D, MEDIAN, NBODY, PI, and SPECTRAL applications have similar values, with a slight advantage for the ILLP.

In Figure 6, one can see the percentage of energy savings for each application using AS and ILLP with $pd = 1$ and $pd = 8$. Figure 6(a) highlights that the ILLP provides better energy reduction values. For instance, the CONV1D application achieved a reduction of 49.11% with ILLP and $pd = 1$ and 11.93% with AS and $pd = 1$.



(a) $pd = 1$. (b) $pd = 8$.

Figure 6. AS and ILLP energy saving percentage for each application.

Figure 6(b) presents the percentage of energy saving with $pd = 8$, which shows similar behavior to the instructions and cycle metrics. The AS and ILLP results were quite akin in 6 out of 13 applications, whereas ILLP achieved better results in the other 6 applications.

Figure 7 presents the boxplot for the reduction in instructions, cycles, and energy metrics, in both techniques, for all pd values $(1, 2, 4, 8)$. Each column presents the results by ordering the lowest to the highest value and organizing the data into quartiles and median. The results indicate that the ILLP outperforms the AS in all metrics and pd. For instance, with a $pd = 1$, ILLP presents an average instruction reduction of 22.55%, whereas the AS only shows a reduction of -1.52%. This trend continues for cycle and energy, where ILLP reaches a reduction of 23% with $pd = 2$ in the number of cycles. However, it is important to note that most results with $pd = 8$ are the same for both techniques, indicating that this pd may be an upper bound for general performance gains and energy savings.

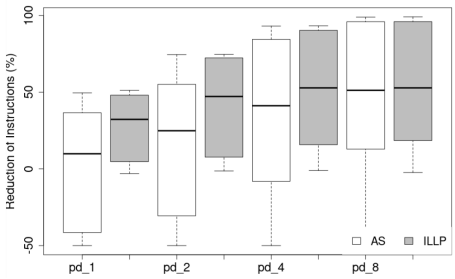
7. Conclusions

This work presented a new technique named Instruction-Level Loop Perforation (ILLP). The approach replaces the loop-iteration operation with an approximate instruction-level operation. ILLP impacts the application's final performance once it skips loop iterations by applying faster instructions to calculate the loop step.

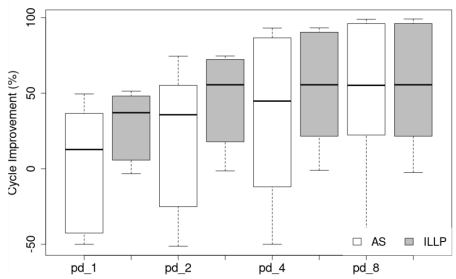
The proposed technique was carried out on the ACCEPT and the SPIKE. The ACCEPT was instrumented to replace original user-oriented loops with the ILLP technique. The application code is then converted into RISC-V instructions. The SPIKE was extended (SPIKEX) to support new approximate instructions following the RISC-V format.

This work carried out experiments on accuracy (RE), number of instructions, number of cycles, and energy (μWsec). The proposed technique (ILLP), the LP in the ACCEPT (AS), and the baseline code were evaluated in a set of thirteen applications. The ILLP and AS experiments were organized into four pd.

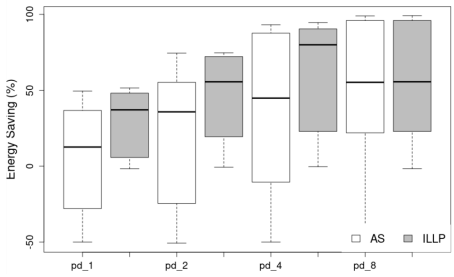
The results brought from the ILLP has proven to be effective in improving several aspects of performance while maintaining accuracy levels equal to the AS. Specifically, the ILLP approach was able to achieve a significant reduction in the number of instruc-



(a) Reduction of Instructions.



(b) Cycle Improvement.



(c) Energy Saving.

Figure 7. Percentage reduction for instructions, cycles, and energy by technique.

tions, cycles, and energy consumption for various applications compared to the BL. For example, the PI application with a $pd = 2$ achieves a 74.61% reduction in the number of instructions, while the FFT application with $pd = 1$ showed a 51.40% reduction in the number of cycles. The PI application using ILLP and $pd = 2$ achieving 74.49% energy saving compared to the original baseline code.

This work can be extended by applying the ILLP in a larger number of loops, replacing other loop-iterations operations, and adopting new approximate instructions, such as $subx$, $mulx$, and $divx$. Another opportunity for future research lies in exploiting approximate instructions in time-consuming applications with mathematical functions, such as sine, cosine, power, and logarithm. The ILLP is not just limited to static compilation, but it can also be applied to dynamic runtime environments such as virtual machines, runtime monitors, and dynamic binary translators. Those environments can improve application performance and reduce energy consumption at the cost of controlled accuracy loss by dynamically replacing accurate instructions with approximate ones.

Acknowledgements

The authors thank Brazilian Research Agencies FUNDECT, CAPES, and CNPq, and UFMS for their financial support to the Research Laboratory of High Performance Computing Systems (LSCAD). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

Almurib, H. A. F., Kumar, T. N., and Lombardi, F. (2016). Inexact designs for approximate low power addition by cell replacement. In Conference on Design, Automation & Test in Europe, page 660–665, CA, USA. EDA Consortium.

Boyini, K. (2022). Floyd warshall algorithm. https://www.tutorialspoint.com/Floyd-Warshall-Algorithm.

Catelan, D., Santos, R., and Duenha, L. (2022). Evaluation and characterization of approximate arithmetic circuits. Concurrency and Computation: Practice and Experience, page e6865.

Game, B. (2022). https://benchmarksgame-team.pages.debian.net/benchmarksgame.

GeeksforGeeks (2022). Dijkstra's shortest path algorithm. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. In Symposium on Compiler Construction, page 120–126, NY, USA. Association for Computing Machinery.

Li, S., Park, S., and Mahlke, S. (2018). Sculptor: Flexible approximation with selective dynamic loop perforation. In International Conference on Supercomputing, page 341–351, NY, USA. Association for Computing Machinery.

Moreno, A. A., Calle, F. R., and Pedraza, C. (2021). A low-cost fault tolerance method for arm and risc-v microprocessor-based systems using temporal redundancy and approximate computing through simplified iterations. Journal of Integrated Circuits and Systems, 16(3).

Reis, L. and Wanner, L. (2021). Functional approximation and approximate parallelization with the accept compiler. In IEEE 33rd International Symposium on Computer Architecture and High Performance Computing, pages 188–197.

Reis, L. O. P. (2021). Targeting broad software approximations with the accept compiler.

RISC-V (2022). Risc-v toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain.

Rodriguez-Cancio, M., Combemale, B., and Baudry, B. (2019). Approximate loop unrolling. page 94–105, NY, USA. Association for Computing Machinery.

Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L., and Oskin, M. (2015). Accept: A programmer-guided compiler framework for practical approximate computing. University of Washington Technical Report UW-CSE-15-01, 1:1–14.

Santos, F. (2022). Cfast fourier transform. https://github.com/riscv-software-src/riscv-isa-sim.

Sidirogloy-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. Association for Computing Machinery, pages 124–134.

Silveira, J., Castro, L., Araújo, V., Zeli, R., Lazari, D., Guedes, M., Azevedo, R., and Wanner, L. (2022). Prof5: A risc-v profiler tool. In International Symposium on Computer Architecture and High Performance Computing, pages 201–210.

SPIKE (2019). Spike risc-v isa simulator. https://github.com/riscv-software-src/riscv-isa-sim.

Statescu, A. (2022). Program to compute pi using a monte carlo method. https://gist.github.com/thinkphp/0d56dfd5eb5f91da029a91d4c7676f12.