Introducing software pipelining for the A64FX processor into LLVM

Masaki Arai
Fujitsu Limited
Kawasaki, Japan
arai.masaki@fujitsu.com

Naoto Fukumoto
Fujitsu Limited
Kawasaki, Japan
fukumoto.naoto@fujitsu.com

Hitoshi Murai
RIKEN R-CCS
Kobe, Japan
h-murai@riken.jp

ABSTRACT

Software pipelining is an essential optimization for accelerating High-Performance Computing(HPC) applications on CPUs. Modern CPUs achieve high performance through many-core and wide SIMD instructions. Software pipelining is an optimization that promotes further performance improvement of HPC applications by cooperating with these functions. Although open source compilers such as GCC and LLVM have implemented software pipelining, it is underutilized for the AArch64 architecture. We have implemented software pipelining for the A64FX processor on LLVM to improve this situation. This paper describes the details of this implementation. We also confirmed that our implementation improves the performance of several benchmark programs.

CCS CONCEPTS

• **Software and its engineering → Retargetable compilers**.

KEYWORDS

compiler, optimization, software pipelining, LLVM, AArch64, A64FX

ACM Reference Format:
Masaki Arai, Naoto Fukumoto, and Hitoshi Murai. 2024. Introducing software pipelining for the A64FX processor into LLVM. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCAsiaWS 2024), January 25–27, 2024, Nagoya, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3636480.3637093

1 INTRODUCTION

Software pipelining is an essential optimization for accelerating High-Performance Computing(HPC) applications on CPUs. Modern CPUs achieve high performance through many-core and wide SIMD instructions. Software pipelining is an optimization that promotes further performance improvement of HPC applications by cooperating with these functions. Although open source compilers such as GCC[14] and LLVM[7] have implemented software pipelining, it is underutilized for the AArch64 architecture.

We have implemented software pipelining for the A64FX processor[9] on LLVM to improve this situation. The A64FX is an out-of-order superscalar processor designed for HPC, compliant with the ARMv8-A architecture profile. The A64FX supports the Scalable Vector Extension (SVE) instructions, a vector extension of the ARM instruction set architecture. The A64FX supports 128, 256, and 512-bit SVE vector lengths.

There have been many previous studies on software pipelining[1], and comprehensive data on the relationship between various algorithms and CPU architecture exist. However, the A64FX differs from the CPUs considered in these studies in the following ways:

- It achieves high performance by utilizing wide SIMD instructions.
- The instructions have an overall long latency.
- It has a complicated execution flow that divides one architectural instruction into multiple micro-operation instructions and executes them.
- Under various conditions, additional penalty cycles occur during these micro-operation instructions.

These features of the A64FX require additional efforts to implement software pipelining.

This paper describes our current work on introducing software pipelining for the A64FX into LLVM. The rest of the paper is organized as follows. Section 2 describes the current status and issues of software pipelining for LLVM. Section 3 describes the details of our implementation of software pipelining for the A64FX. Section 4 describes performance evaluation. Section 5 describes future work. Section 6 provides conclusions.

2 CURRENT STATUS AND ISSUES OF LLVM

LLVM 17 has a MachinePipeliner pass as an optimization pass to perform software pipelining. The architectures using the MachinePipeliner pass are ARM, Hexagon, and PowerPC, and AArch64 is not supported. The algorithm used by MachinePipeliner is the Swing Modulo Scheduling(SMS)[10] algorithm, characterized by short-time optimization and register-constraint-aware kernel generation. However, some previous research[3] has shown that the Iterated Modulo Scheduling(IMS)[13] algorithm produces better results than SMS for complex architectures.

The MachinePipeliner pass applies the optimization in Static Single Assignment(SSA) form[4] and maintains the SSA form for the instruction sequence generated by the optimization results. For this reason, the MachinePipeliner pass can use the subsequent register allocation pass without modification after its optimization. The problem with this implementation method is that it requires many PHI instructions to maintain the SSA form after the optimization, and these PHI instructions become register-to-register COPY instructions as a result of non-SSA conversion and remain as COPY instructions after register allocation, which may damage the schedule results. Furthermore, because the MachinePipeliner pass and the

HPCAsiaWS 2024, January 25–27, 2024, Nagoya, Japan

ACM ISBN 979-8-4007-1652-2/24/01…$15.00
https://doi.org/10.1145/3636480.3637093

register allocation pass are independent, the MachinePipeliner pass cannot guarantee that the register allocation pass will not generate spill code in the kernel part generated as a result of its optimization. In addition, although it is important for software pipelining to cope with the shortage of architectural registers, for example, to alleviate register pressure by Stage Scheduling[5] and spill code generation[16], the MachinePipeliner does not currently implement these functions. Moreover, MachinePipeliner does not perform Modulo Variable Expansion(MVE)[6]. As a result, MachinePipeliner has the advantage of being able to reduce the code size of its optimization results. The trade-off is that the optimized kernel will retain register-to-register COPY instructions, and these instructions may cause performance degradation.

Detailed analysis of loop dependency distances, loop unrolling, loop distribution/fusion, and other coordinated operations to adjust loop sizes and instruction types enhance the optimization effects of software pipelining. However, LLVM does not realize the cooperation of these functions.

Software pipelining can optimize various innermost loops, but when applied to loops with a small number of executions, the overhead of executing additional preprocessing and postprocessing code can adversely affect performance. It is also desirable to apply software pipelining after unrolling the loop an appropriate number of times, considering the instruction dependencies inside the loop and the usage status of hardware resources. Since it is generally difficult for the compiler to make these judgments automatically at compile time, a mechanism that allows the user to give optimization instructions by specifying directives in the source code would be useful, but LLVM does not currently have such a feature.

The description processed by LLVM's TableGen program can express in detail the scheduling model for the instructions for the target CPU. This description specification can describe the division of an architectural instruction into multiple micro-operations. However, the current description specification cannot specify the pipeline that executes each divided micro-operation or express changes in latency due to dependencies between micro-operations.

3 SOFTWARE PIPELINING IMPLEMENTATION FOR THE A64FX

This section describes the details of our implementation of software pipelining for the A64FX processor on LLVM. Our implementation of software pipelining for the A64FX has the following features:

- We adopt Iterated Modulo Scheduling as a scheduling algorithm.
- Our implementation extends the LLVM scheduling model for the A64FX.
- We perform instruction scheduling in non-SSA form.
- Our implementation applies Modulo Variable Expansion if necessary due to instruction scheduling.
- We perform register allocation to the kernel part after instruction scheduling.
- We have introduced countermeasures for register shortages due to register allocation.

Algorithm 1 shows an overview of our implementation's processing flow within the software pipelining pass. Our implementation applies software pipelining to a loop consisting of a single basic block, resulting in the control flow graph shown in Figure 1. Implementation details are described in the following sections.

Algorithm 1: An overview of our implementation's processing flow within the software pipelining pass

```
INPUT  : Basic Block BB
OUTPUT : {SUCCESS, FAILURE}
Perform various preprocessing steps
Convert BB to non-SSA form
II ← Calculate minimun II
regshort ← false
genspill ← false
while true do
    II ← Execute Iterated Modulo Scheduling (II)
    if II does not improve the performance of the original loop then
        if regshort == false or genspill == true then
            Give up applying software pipelining
            return FAILURE
        Introduce spill code to relieve register pressure
        genspill ← true
        II ← Recalculate minimun II
        continue
    Execute Stage Scheduling
    Apply Modulo Variable Expansion if necessary
    Allocate registers to the kernel part
    if Register allocation was successful then
        break
    else if The failure was due to instruction format constraints then
        Insert COPY instructions
        II ← Recalculate minimun II
    else
        regshort ← true
        II ← II + 1
Generate the resulting optimized code including the extra loop
Replace BB with the optimized code and update CFG
return SUCCESS
```

3.1 Various pre-processing

As a preprocessing before applying software pipelining, we analyze the basic blocks to be optimized, detect the number of loop rotations, and rewrite them into a form that is easy to apply software pipelining.

In addition, our implementation decomposes the pre and post-indexed instructions of the AArch64 instruction set into load or store instructions and address addition instructions at this stage to increase the flexibility of scheduling those instructions.

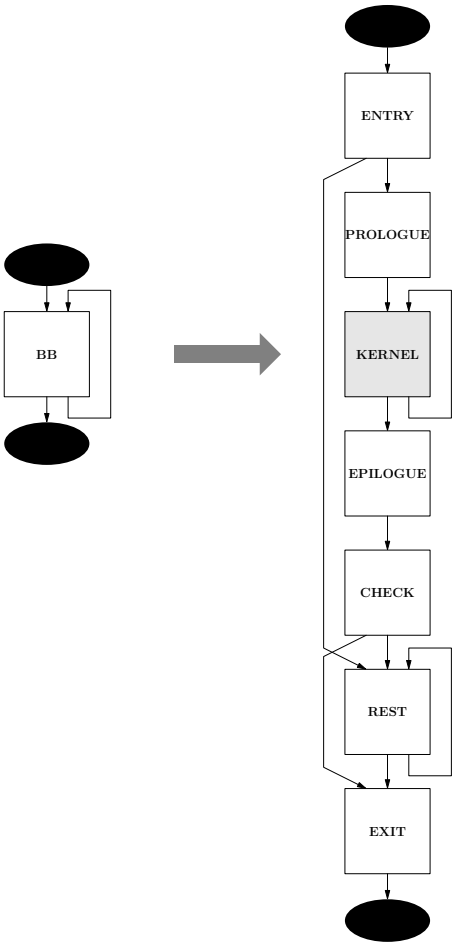

Figure 1: Control flow graph of optimization results

3.2 Converting non-SSA form

To prevent scheduling results from being corrupted by COPY instructions generated from LLVM's non-SSA transformation[2], our implementation converts the basic block to non-SSA form before optimization. With this, we delete PHI instructions, create an instruction sequence that includes the minimum necessary COPY instructions, and execute instruction scheduling. Although there is room for selection in the position to generate the COPY instructions required when deleting PHI instructions, in our implementation, at this point, we are minimizing the number of COPY instructions that appear in the instruction sequence that we are scheduling. We convert only the KERNEL part in Figure 1 to the non-SSA form, and for other basic blocks, by maintaining the properties of the SSA form, we can use the subsequent PHI deletion and register allocation pass of LLVM. As a result of the non-SSA conversion at this stage, we can only deal with architectural instructions that are finally output as assembly code during instruction scheduling.

In addition to the PHI and COPY instructions, which are pseudo instructions related to the SSA form, LLVM uses subregister-related instructions as pseudo instructions that express restrictions for register usage(`INSERT_SUBREG`, `SUBREG_TO_REG`,`REG_SEQUENCE`). Our implementation removes these subregister-related pseudo instructions after saving their constraint information needed for register allocation at this stage and avoids handling them during instruction scheduling.

3.3 Iterated Modulo Scheduling

We use IMS algorithm as the software pipelining algorithm in our implementation. Some previous research[3] has shown that the IMS algorithm produces better results than SMS for complex architectures. The A64FX is a more complex CPU than the complex architectural model used in this paper. The A64FX has longer instruction latency than its complex architectural model, and the method of executing instructions is also more complicated. The A64FX has a complex implementation that divides one instruction into multiple μOP instructions and puts each one into the available pipeline for execution.

For example, we explain the schedule model of architecture instruction LD2W (scalar plus scalar) in the A64FX. The A64FX decomposes this instruction into μOP instructions for one address calculation (μOP0) and two data loads (μOP1 and μOP2), and it submits the operation to either the EAGA or EAGB pipeline (Architecture Manual [9] Section 16). Therefore, even if there is no effect of other instructions, there are eight execution patterns for this instruction, as shown in Table 1. The address calculation has a latency of 1 cycle, and the load of the two data takes the address value of the calculation result as an input and has a latency of 10 cycles. Figure 2 shows the pipeline flow when the EAGA pipeline performs address calculations, and the EAGA and EAGB pipelines each perform one load. Executing the pattern in Figure 2 has a total latency of 11 cycles. Figure 3 shows the pipeline flow when the EAGB pipeline performs address calculations and the EAGA pipeline performs both loads. Since the EAGA pipeline cannot execute two load instructions simultaneously, an additional one-cycle delay will occur in that case, as shown in Figure 3. Since our implementation does not use pipeline execution patterns that cause delays as a scheduling model for instruction scheduling, we only use the four patterns with a latency of 11 cycles in Table 1.

Our IMS implementation considers the registration status of the Modulo Reservation Table and the status of instructions before and after data dependence and uses an appropriate pattern from these patterns to search for solutions. Since the current schedule model description of LLVM cannot express such a complex situation, our implementation stores this information as code in the target-dependent part used by the optimization pass.

After running IMS, we apply Stage Scheduling to reduce register usage. After determining the code for the kernel part as a result of applying Stage Scheduling, we use MVE if necessary.

3.4 Register allocation to the kernel part

As a result of applying IMS, Stage Scheduling, and MVE, our implementation generates code in the form of a control flow graph shown in Figure 1, which consists of a prologue, kernel, and epilogue parts. In Figure 1, KERNEL is a steady-state kernel loop generated by software pipelining, and there are codes for the PROLOGUE and

		0	1	2	3	4	5	6	7	8	9	10	11	12	13
μOP0	EAGA	X	U	UT											
μOP1	EAGA		X/A	T	M	B	XT	XM	XB	R	RT	RT2	RT3/C	W	W2
μOP2	EAGB		X/A	T	M	B	XT	XM	XB	R	RT	RT2	RT3/C	W	W2

Figure 2: Pipeline execution pattern 4 of LD2W (scalar plus scalar) instruction

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
μOP0	EAGB	X	U	UT												
μOP1	EAGA		X/A	T	M	B	XT	XM	XB	R	RT	RT2	RT3/C	W	W2	
μOP2	EAGA			X/A	T	M	B	XT	XM	XB	R	RT	RT2	RT3/C	W	W2

Figure 3: Pipeline execution pattern 1 of LD2W (scalar plus scalar) instruction

Table 1: Pipeline execution patterns of LD2W (scalar plus scalar) instruction

pattern	μOP0	μOP1	μOP2	latency
0	EAGA	EAGA	EAGA	12
1	EAGB	EAGA	EAGA	12
2	EAGA	EAGB	EAGA	11
3	EAGB	EAGB	EAGA	11
4	EAGA	EAGA	EAGB	11
5	EAGB	EAGA	EAGB	11
6	EAGA	EAGB	EAGB	12
7	EAGB	EAGB	EAGB	12

EPILOGUE parts before and after it. If the number of rotations of the original loop is greater than the minimum number of loop body executions on the path that executes KERNEL, the ENTRY basic block selects the path to execute KERNEL. Otherwise, it executes the loop body with the REST equivalent of the original loop body. The basic block of CHECK chooses to execute the remaining iterations by the loop in REST if the number of loop iterations executed in the path including KERNEL is less than the number of rotations of the original loop.

Our implementation generates the KERNEL with non-SSA form as a result of scheduling and allocates registers to it after scheduling. It retains the SSA form for the rest of the basic blocks and utilizes subsequent optimization passes of LLVM for optimization and register allocation of those basic blocks. The LLVM intermediate language specification does not allow hardware registers to live across basic block boundaries on the control flow graph before register allocation. However, it is necessary to use hardware registers to input and output data to and from the KERNEL part. Therefore, in our implementation, we use pseudo instructions to represent the lifetime of hardware registers for value passing through hardware registers. Figure 4 shows an example of the representation of the lifetime of hardware registers using these pseudo instructions. In this Figure, the `LIVE_IN` and `LIVE_OUT` pseudo instructions express that registers `$x<R>` and `$x<W>` live across basic block boundaries. Our implementation removes these pseudo instructions after LLVM register allocation.

The current implementation attempts to reduce register usage by increasing the II value if there is a shortage of architectural registers when allocating registers to the KERNEL part. If increasing the II value does not solve the register shortage, or if increasing the II value eliminates the optimization effect, our implementation introduces spill code[16] to ease the register usage. If introducing spill code does not solve the register shortage, our implementation gives up applying software pipelining to the target loop and restores the original code before optimization.

Even if there is room in the architecture registers when allocating registers to the KERNEL part, it may not be possible to allocate registers due to the restrictions of the instruction format of the AArch64 architecture. If the register allocation fails due to such restrictions on the instruction format, insert a COPY instruction to avoid the conditions and re-execute IMS.

```
PROLOGUE:
  ... write $x<R> ...
  ...
  LIVE_OUT, implicit $x<R>
KERNEL:
  LIVE_IN, implicit-def $x<R>
  ...
  ... read $x<R> ...
  ... write $x<W> ...
  ...
  LIVE_OUT, implicit $x<W>
  Bcc 1, KERNEL
  B EPILOGUE
EPILOGUE:
  LIVE_IN, implicit-def $x<W>
  ...
  ... read $x<W> ...
```

Figure 4: Liverange representation for hardware registers

Table 2: Evaluation Environment

System	PRIMEHPC FX700
CPU	A64FX, 2.0 GHz, 48 cores
Memory	32GiB(HBM2)
OS	CentOS 8.3
Compiler	LLVM 17.0.4
Compile option	-O2 -fno-unroll-loops -msve-vector-bits=512

3.5 Cooperation with other optimization passes

Our current implementation does not apply loop unrolling to loops subject to software pipelining. When software pipelining optimizes loops after applying loop unrolling, register allocation to the kernel part often fails due to a lack of registers. In addition, we do not apply the instruction scheduling or the peephole optimizations after register allocation to kernel loops that apply software pipelining. The reason is that those optimizations to the kernel loop may destroy the execution timing by IMS and cause performance degradation.

4 PERFORMANCE EVALUATION

This section describes the performance evaluation of our implementation of the software pipelining pass. Table 2 shows our evaluation environment. We used the Livermorec[12] benchmark and the TSVC [11] benchmark as benchmark programs for evaluation. For the evaluation, we measured only the innermost loop of each benchmark program and set the number of loop revolutions to 32×10^6. For all benchmarks, we rewrite loops that can be executed in parallel into SIMD executable code using the ACLE descriptions[8]. Loop unrolling was not applied to avoid register shortage situations. Furthermore, accurate data dependence distance information is provided at compile time via command line options for all benchmarks if necessary, Table 3 shows the results of the performance evaluation. In Table 3, the noswpl and swpl columns show the execution time without and with software pipelining, respectively. The speedup column shows the performance improvement due to software pipelining. These values confirm that for each kernel, the application of software pipelining improves performance. Kernel 3 includes an instruction FADDA with a long latency of 69 cycles in the loop, so it cannot take advantage of software pipelining. TSVC s273 shows that the reordering of instructions by software pipelining is more effective than reordering instructions by hardware at runtime. In Table 3, the II column shows the value of the initiation interval resulting from modulo scheduling, the stages column indicates the number of stages in the result, and the unroll column shows the number of times MVE unrolled the loop body. From the values in the MVE column, we can see that it is necessary to use MVE to generate high-performance kernels. In Table 3, the II_∞ column represents the initiation interval value, assuming an infinite number of registers exist. The difference between the II_∞ and II column values indicates that the lack of registers suppresses instruction-level parallelism in most benchmarks.

Table 3: Evaluation Results

benchmark	noswpl(μs)	swpl(μs)	speedup	II_∞	II	stages	unroll
kernel 1	17490	15666	1.12	4	6	8	8
kernel 2	32122	18110	1.77	4	7	7	6
kernel 3	147642	147640	1.00	109	109	1	1
kernel 5	291496	291193	1.00	20	20	2	3
kernel 7	29308	29783	0.98	6	12	5	6
kernel 11	146160	145905	1.00	10	10	3	3
kernel 12	12497	12370	1.01	2	3	8	8
TSVC s1161	47819	24994	1.91	6	11	4	5
TSVC s271	10213	8852	1.15	3	7	6	7
TSVC s272	46282	25662	1.80	5	7	7	7
TSVC s273	152535	37628	4.05	7	8	7	8
TSVC s274	31210	25389	1.23	6	10	6	7
TSVC s1279	33708	26696	1.26	4	10	6	7
TSVC s2712	6690	6068	1.10	3	7	6	7
TSVC s4113	54028	34247	1.58	23	23	4	4

5 FUTURE WORK

Because of the long instruction latency of the A64FX, the number of stages tends to increase when Modulo Scheduling is applied, resulting in high register pressure. For this reason, we are considering introducing a method to reduce register pressure during scheduling and a process to alleviate register pressure by introducing spill code. Since it is difficult to improve register pressure only by a software pipelining pass, adjusting the loop size by loop distribution/fusion and loop unrolling is necessary. Implementing these methods and evaluating their impact on performance will be the subject of future work.

There are if-conversion and reverse-if-conversion techniques for applying software pipelining to loops containing if statements. SVE's predicate can convert an if statement into a mask process for vectorized loops, making the code applicable to software pipelining. However, the SVE instruction requires using hardware resources and the definition of registers even when the predicate data is all false. Therefore, it may be better to generate code that does not consume hardware resources by applying reverse-if-conversion rather than converting unbalanced if statements to the predicate form. We are considering introducing hierarchical reduction[6] and reverse-if-conversion[15] for loops of scalar instructions and implementing the decision to apply vectorization from the perspective of software pipelining.

In the current LLVM, the autovectorizer converts a loop with scalar instructions into a vectorized loop using SVE instructions. This process generally generates pre-processing and post-processing code before and after the loop targeted for optimization to adjust the number of loop executions and memory alignment. We perform software pipelining after these automatic vectorizations. For this reason, our pass generates the code shown in Figure 1 in the form of further additions to this pre-processing and post-processing, resulting in the expansion of code size due to the execution of redundant branch instructions and a series of loops with a small number of executions. Reducing these redundant codes by coordinating the software pipelining pass and the automatic vectorization function is future work.

6 CONCLUSIONS

Software pipelining is an essential optimization for accelerating HPC applications on CPUs. This paper details the implementation of software pipelining for the A64FX processor on LLVM and evaluates its performance. The A64FX is an out-of-order type superscalar processor that divides an architectural instruction into multiple micro-operation instructions and submits each instruction to an execution pipeline. Although the A64FX is a CPU with a complex execution flow, we confirmed that our implementation improves the performance of several benchmark programs. In the future, we plan to study how to deal with register shortages and coordinate our software pipelining with the overall flow of LLVM's optimization pass. We are also considering proposing our implementation to LLVM upstream.

ACKNOWLEDGMENTS

We want to thank the developers at Metro Corporation for their help with the implementation.

REFERENCES

[1] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. 1995. Software Pipelining. *ACM Comput. Surv.* 27, 3 (sep 1995), 367–432. https://doi.org/10.1145/212094.212131

[2] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. 2009. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *2009 International Symposium on Code Generation and Optimization*. 114–125. https://doi.org/10.1109/CGO.2009.19

[3] Josep M. Codina, Josep Llosa, and Antonio González. 2002. A Comparative Study of modulo Scheduling Techniques. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, USA) *(ICS '02)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/514191.514208

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[5] A.E. Eichenberger and E.S. Davidson. 1995. Stage scheduling: a technique to reduce the register requirements of a module schedule. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 338–349. https://doi.org/10.1109/MICRO.1995.476843

[6] M. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 318–328. https://doi.org/10.1145/53990.54022

[7] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[8] ARM Limited. 2015. Arm C Language Extensions. https://arm-software.github.io/acle/main/acle.html Accessed: Nov. 13, 2023.

[9] Fujitsu Limited. 2022. A64FX Microarchitecture Manual v1.8.1. https://github.com/fujitsu/A64FX Accessed: Dec. 1, 2022.

[10] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. 1996. Swing module scheduling: a lifetime-sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. 80–86. https://doi.org/10.1109/PACT.1996.554030

[11] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 372–382. https://doi.org/10.1109/PACT.2011.68

[12] Tim Peters. 1992. Livermore Loops coded in C. https://netlib.org/benchmark/livermorec Accessed: Dec. 1, 2022.

[13] B. Ramakrishna Rau. 1994. Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose, California, USA) *(MICRO 27)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/192724.192731

[14] Richard M Stallman et al. 1999. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation Boston, MA, USA.

[15] Nancy J Warter, Grant E Haab, Krishna Subramanian, and John W Bockhaus. 1992. Enhanced modulo scheduling for loops with conditional branches. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 170–179.

[16] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. 2000. Improved Spill Code Generation for Software Pipelined Loops. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 134–144. https://doi.org/10.1145/349299.349319