# A Case for Hardware Memoization in Server CPUs

Farid Samandi, Natheesan Ratnasegar, and Michael Ferdman

***Abstract*—Server applications exhibit a high degree of code repetition because they handle many similar requests. In turn, repeated execution of the same code, often with identical inputs, highlights an inefficiency in the execution of server software and suggests memoization as a way to improve performance. Memoization has been extensively explored in software, and several hardware- and hardware-assisted memoization schemes have been proposed in the literature. However, these works targeted memoization of mathematical or algorithmic processing, whereas server applications call for a different approach. We observe that the opportunity for memoization in servers arises not from eliminating the repetition of complex computation, but from eliminating the repetition of software orchestration code. This work studies hardware memoization in servers, ultimately focusing on one pattern, instruction sequences starting with indirect jumps. We explore how an out-of-order pipeline can be extended to support memoization of these instruction sequences, demonstrating the potential of hardware memoization for servers. Using 26 applications to make our case (3 CloudSuite workloads and 23 vSwarm serverless functions), we show how targeting just this one pattern of instruction sequences can memoize over 10% (up to 15.6%) of the dynamically executed instructions in these server applications.**

***Index Terms*—Microarchitecture, hardware memoization.**

## I. Introduction

SERVERS may handle thousands of operations per second, comprising a continuous stream of independent user requests. Although the behavior of the same server software varies drastically across sites, the variety of requests that a given server handles is limited by its environment. For example, while web servers at different sites serve completely different mixes of requests and different content, any particular web server receives a relatively small diversity of requests to its popular pages. Similarly, database installations differ radically, but a given database will handle only a small variety of SQL queries. This behavior inevitably leads to high cross-request instruction stream similarity, resulting in server processors repeatedly executing the same instruction sequences, with exactly the same input register values (and therefore exactly the same output register values).

Frequent repetition of the same instruction sequences presents an opportunity to improve server performance through memoization. Primarily a software technique, memoization has been applied at coarse granularity for algorithmic efficiency (e.g., accelerating minimax with alpha-beta pruning to avoid redundant evaluations of game positions) and for system optimization (e.g., caching of path lookups in filesystem code). At finer granularity, several hardware proposals for automated memoization explored the possibility of freeing up functional units by avoiding the re-execution of instructions with identical inputs [1], [2]. These approaches retrieve register results computed by a previous execution of the ALU operation on the same inputs, thereby freeing resources (primarily ALUs) to work on other instructions.

In this work, we consider automated hardware memoization for servers. Counterintuitively, although memoization has traditionally been used to avoid repeated computation, we see that the dominant memoization opportunity in servers is not avoiding computation, but rather in memoizing instructions responsible for software orchestration. Server software is characterized by large complex code bases with a high density of control-flow and non-arithmetic instructions, primarily performing data lookups and copies. As such, memoization opportunities in servers arise in the spaces between computation, avoiding repeated execution of mundane instruction sequences that adapt (and verify the validity of) parameter values as execution progresses through deeply nested API hierarchies and software abstractions. Notably, such spaces between computation comprise the bulk of the instructions that modern servers execute.

To evaluate the potential of hardware memoization, we created an infrastructure for studying full-system server applications, using a network-connected RISC-V system with an out-of-order core running on an FPGA, booting Linux and running off-the-shelf server software in Docker containers. We use this system to collect a detailed instruction-level trace of server applications while they service requests arriving over the network interface.

Analysis of our server traces points to significant opportunity for hardware memoization, but also highlights many challenges. To offer some examples: Server applications are large and do not have dominant patterns, suggesting that a generic mechanism to learn and memoize sequences may be cost-prohibitive or even impossible. Compiler support may enable longer sequences to be memoized or provide hints of where memoization opportunities occur, but will require work on compilers and ISA extensions. Whereas system calls and floating point instructions are poor memoization candidates, memoizing instruction sequences with load instructions that access static data structures may be profitable, but would incur significant hardware costs for value tracking (e.g., through the on-chip coherence mechanisms).

Recognizing that a single work cannot cover all cases, we select one example design point to make the case that hardware memoization in servers deserves further research. Specifically, statistical analysis of our traces finds that many memoization opportunities exist for sequences of instructions that begin on an indirect jump, and that the cost of memoizing these sequences is lower than executing them, suggesting potential performance benefits. These candidate sequences comprise almost 10% of the instructions executed by our server applications. The mechanisms we envision extend the existing branch prediction and branch resolution hardware to add support for: (1) speculatively identifying when a memoization candidate sequence is encountered, (2) performing memoization by substituting the previously recorded values into the output registers, (3) validating the input register values, and (4) rolling back in case input validation fails.

In the rest of this paper, we describe our hardware memoization design for servers, compare it to prior work, and present an analysis of 26 server applications, projecting the behavior and approximate on-chip storage costs of this technique.

## II. History of Memoization

The concept of *memoization* was proposed by Michie [3], recognizing that once a computer has executed a computation, the results can be stored to avoid repeating the same computation. Since then, software memoization was explored at various levels of granularity [4], [5], [6], including hardware and ISA extensions to support finer-grained memoization [7], [8], [9], [10]. These works rely on the observation that computation frequently revisits the same sub-problems, creating

Received 18 April 2024; revised 9 October 2024; accepted 20 November 2024. Date of publication 22 November 2024; date of current version 3 December 2024. *(Corresponding author: Natheesan Ratnasegar.)*

The authors are with the Department of Computer Science, Stony Brook University, Stony Brook, NY 11794 USA (e-mail: fsamandi@cs.stonybrook.edu; nratnasegar@cs.stonybrook.edu; mferdman@cs.stonybrook.edu).

Digital Object Identifier 10.1109/LCA.2024.3505075

an opportunity to store the sub-problem results and retrieve them when the same sub-problems repeat.

Unlike complex computational workloads, server applications are primarily tasked with accessing and moving data. Despite this, modern server applications make heavy use of software memoization, although it is typically referred to as *caching*. Servers cache filesystem objects and database queries to avoid repeated traversal of large data structures, especially when these data structures are kept in slower storage. In all cases, whether for complex computation or for caching of simple objects, the software developer, potentially assisted by development tools, identifies potential opportunities and modifies the software to memoize results instead of recomputing or reloading them.

When we examine the execution of optimized server applications (which have caching applied by the software developers where appropriate), we see that there still remains significant opportunity for memoization. Server software is typically developed by many engineers and comprises large code bases with many interworking components; even if applications have a single developer, they rely heavily on runtimes, libraries, and the operating system, developed by others. In this environment, modularity and composability of components are prioritized, with a large fraction of the execution time responsible for the glue logic and wrappers between the various components. Some of this glue logic is built automatically, such as the linkage tables of dynamically linked libraries [11] and the dispatch tables of virtual function calls and overloaded operators, while others are explicitly put in by the developers, such as parameter shuffling where arguments to a function or its return values are rearranged before being passed to further function calls. These software constructs execute instruction sequences that frequently repeat with the same register inputs, as they are not dependent on any input from the requests being served.

Prior works in hardware memoization considered instruction-level memoization [1] and the memoization of short instruction sequences [2], [12], but were primarily concerned with reducing the pressure on the functional units in computational workloads, rather than targeting memoization of sequences with control flow instructions. However, memoizing repeating instruction sequences in servers poses a different set of challenges compared to traditional memoization or caching. First, because the repeating sequences are relatively short, the number of instructions needed to look up and reuse stored values may exceed the original sequence being memoized, suggesting the need for hardware support. Second, the repeating sequences often include control flow instructions that span function boundaries, where a sequence may begin in the middle of one function and end in the middle of another, necessitating memoization at the instruction level. Performing memoization in hardware lends itself well to such cases, as it enables the hardware to cost-effectively monitor the dynamic instruction stream to identify and leverage the opportunities presented by these instruction sequences.

## III. Hardware Memoization in Server CPUs

In this work, we study the instruction traces of a collection of server applications to understand the available memoization opportunities. For each application, we identify memoization candidates by finding sequences of eligible consecutive instructions that repeat with identical register input values. We classify syscalls, fences, atomics, and instructions manipulating control registers as ineligible for memoization, as these instructions have complex side-effects. We treat floating point instructions as ineligible for memoization because they are infrequent in server software. Although load instructions can be memoized, for our simple hardware design, we treat loads as ineligible, leaving memoization of sequences containing loads for future work.

Due to the large size and variety of functionality of server software, we found that no particular instruction sequence dominates in any application. Unlike computationally-intensive software, where a small number of tight loops could represent the vast majority of memoization opportunities, even the most-frequently repeating instruction sequence in servers comprises only a tiny fraction of execution. As a result, instead of trying to determine which instruction sequences are good memoization candidates, we analyze our traces to identify the most common instruction sequence *patterns* that correspond to a significant fraction of memoization candidates.

Each pattern corresponds to many repeating instruction sequences in every server application that we studied. However, even when considering the most common patterns, no pattern stands out as being dominant and contributing to the bulk of the memoization opportunities. The patterns also differ significantly from one another, and research is needed to develop hardware mechanisms that work for a large variety of patterns. Recognizing the challenges in constructing a generalized solution, in this preliminary study, we focus only on the most common instruction sequence pattern that we found, which we refer to as *ID-J sequences*, offering a case study of hardware memoization in servers and motivating further exploration.

ID-J sequences begin with an indirect jump instruction and continue until the first ineligible instruction. Although indirect jumps appear in many scenarios in servers, the most common are function returns. As a result, the most common ID-J instruction sequences start with a function's return instruction and continue at that function's call site, where the return values are either checked for errors or passed as arguments to another function (after potentially being shuffled between registers). ID-J sequences terminate either several instructions after returning to the call site, or in the body of a subsequently called function.

To memoize ID-J sequences, the hardware should detect the start of a previously observed sequence and automatically substitute the *original instruction sequence* with a *replacement sequence* that has the same register outputs and continues execution at the same location where the original sequence ended. The process of substituting and executing the replacement sequence must incur a lower cost than the original sequence, by executing fewer instructions, producing the outputs earlier to satisfy dependencies, using fewer micro-architectural resources, or a combination of these benefits. Finally, an important goal for hardware memoization should be to introduce minimal changes to the complex out-of-order processor and to avoid introducing additional latency to the operations on the critical path.

The constraints and needs of memoizing ID-J sequences lend themselves well to leveraging a number of existing micro-architectural components. Specifically, we envision using the branch predictor, branch recovery logic, and the memory hierarchy to implement hardware memoization. By storing the replacement sequences in memory, we are able to use the existing instruction fetch and cache infrastructure to bring replacement sequences to the frontend and dispatch them into the pipeline. Because our target sequences begin with an indirect jump, the branch predictor is already expected to redirect the program counter to the destination of the jump, which can be modified to instead redirect the program counter to the address of the replacement sequence, essentially "for free" relative to the traditional frontend organization. Injecting the replacement instruction sequence in this way cleanly integrates with the micro-architecture of a modern processor and enables speculative elimination of the original sequence, from fetch to commit.[1]

Notably, this way of introducing hardware memoization into the pipeline leaves the timing-critical fetch path unaffected. Only minor modifications to the semantics of some replacement sequence instructions are needed to treat the commit of these instructions as correct-path execution, or to detect mis-speculation when the input register values do not match the values checked by the replacement sequence. For validation, the replacement sequence reads the same register values as the original sequence, ensuring that any not-yet-resolved values and bypass paths are respected when validating the memoization. Moreover, the most computationally complex mechanism of identifying memoization candidates and writing replacement sequences into memory can operate

[1]We use the term *speculative* to denote fetching a replacement instruction sequence (in place of the original sequence), where the replacement sequence may be rolled back due to mis-speculation. This differs from prior work labeled *Reuse through Speculation on Traces*, which performs value prediction, but still always fetches and decodes the original instruction sequences [13].

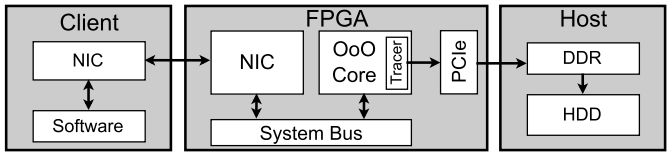


Fig. 1. Trace collection infrastructure.

after the retire stage, off of the critical path. Because server applications execute memoization candidates countless times (sometimes for years of continuous uptime), the training time for the memoization candidates is irrelevant, allowing this operation to be done in hardware over many observations of the candidate sequences, or may even allow future implementations to run complex training algorithms in firmware or software.

## IV. Methodology and Experimental Setup

Studying memoization on servers requires analyzing very long traces, which must be collected without perturbing the timing of the server's execution as it handles requests. Moreover, studying a realistic server system requires support for complex functionality (such as the Linux OS, Docker containers, high-speed NICs, etc.). To maximize the fidelity of our traces, we use a hardware emulator to faithfully model the system. We use Chipyard [14] to generate a full-system SoC with RISC-V cores to run on an FPGA. Our system includes a 3-wide superscalar out-of-order core with 32 KB 8-way set-associative L1-I and L1-D caches, and a unified 512 KB 8-way set-associative L2.

To collect instruction traces, we implemented a hardware trace collector buffer and a PCIe DMA engine. Fig. 1 shows a diagram of our platform. For every instruction committed by the RISC-V core, a record is appended to the buffer containing the program counter and register input values. The trace is compacted in the buffer and transferred over PCIe to the memory of a host computer, which then stores the data on disk.

We trace and study a suite of server applications. We use Data Caching, Media Streaming, and Web Serving from the latest CloudSuite benchmarks, and 23 serverless functions from the vSwarm benchmark suite [15].[2] For ease of presentation, we aggregate the 23 serverless functions according to their runtime language: *Go*, *Node.js*, or *Python*.

For each workload, the server runs on a RISC-V core, while a client emulator runs on a separate high-end machine, which enables one client machine to significantly load the server under test. The FPGA and the client machine are connected with a high-speed Ethernet link, mimicking a production environment where the clients connect to the server over a network. For each workload, we launch the client and wait for the server to reach a steady state before starting to capture the records of committed instructions. We collect 10 billion instructions for each application, comprising minutes of wall-clock time on the RISC-V core and ensuring that the trace includes many complete server requests for all workloads.

## V. Memoization Cost and Benefit Projections

In this section, we first quantify the opportunity of memoization in servers by exploring the repetition of unique dynamic instructions, highlighting the potential benefits of pursuing this direction in future research. Then, as an initial step in this direction, we focus on just one of the patterns that make up memoization opportunity and estimate the performance gains that could be achieved by memoizing the ID-J sequences. Finally, we perform an analysis of the number of ID-J sequences that must be tracked to facilitate memoization, which serves as a first-order cost estimate of the memoization hardware.

[2]We do not trace the other CloudSuite and vSwarm workloads because they require Java, which is not well supported on RISC-V.

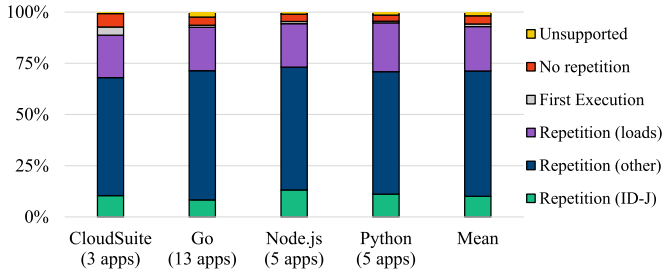


Fig. 2. Breakdown of dynamic instructions repetition.

To quantify memoization opportunity, we process our workload traces to identify all unique dynamic instructions (determined by their program counter and register inputs) that are eligible for memoization, and count their number of repetitions. Fig. 2 shows the breakdown of the instructions, where each bar represents an average across multiple applications written in the same programming language, and the rightmost bar shows the average across all 26 applications. The combination of the bottom three sections of each bar (green, blue, and purple) corresponds to instructions that exactly repeat their execution, having the same program counter, same register input values, and same register output value.

We find that, on average, 93% of the instructions are candidates for hardware memoization, indicating massive opportunity. The bottom three sections of the bars correspond to instructions that are part of ID-J sequences (green), other repeating ALU instructions (blue), and memory loads which load exactly the same value from exactly the same address on each execution (purple). Remarkably, only 1% of the dynamic instructions correspond to the first time when repeating instructions are observed (gray), suggesting that training an effective memoization predictor requires observing only a small fraction of instructions. Across our workloads, we find only 4% of non-repeating instructions (red), characterized by different input register values (or different values loaded from memory) across executions, making them ineligible for memoization.

The bottom portion of the bars (green) demonstrates the opportunity of memoizing just the ID-J sequences using the approach described in Section III. On average, the repetition of ID-J sequences comprises 10% of the executed instructions. The lowest memoization opportunity we see is 5.6% in the Geo service of the Hotel Reservation Suite implemented in Go, while the highest memoization opportunity of 15.6% is in the Authentication workload implemented in Node.js. The cost of memoization stems from executing the replacement sequence instead of the original ID-J sequence. This replacement sequence serves two purposes: (1) validating the current state of the register file against the recorded register values for ongoing speculation, and (2) storing the recorded output values of the original sequence to the register file. We estimate that a typical replacement sequence will comprise 3-5 instructions. Analyzing our traces, we see that the average length of ID-J sequences in our benchmarks is 7.2 instructions, ranging from a minimum average of 6 instructions (for the Recommendation Service of the Hotel Reservation Suite) to a maximum average of 9.1 instructions (for Media Streaming). This result indicates that executing the replacement sequence is likely less costly than executing the original sequence, resulting in potential performance benefits. Although ID-J sequences represent only a small subset of the repeating instructions in our applications, their significant coverage already suggests that integrating hardware memoization for just these sequences into future processor designs could be beneficial. Furthermore, these results strongly support pursuing memoization opportunities beyond ID-J sequences, as other instruction sequence patterns may also be amenable to memoization with incremental changes to the existing micro-architectural structures.

To estimate the hardware cost of storing memoization metadata, we study the coverage achieved by memoizing ID-J sequences within a constrained storage budget. We sort all ID-J sequences by their coverage

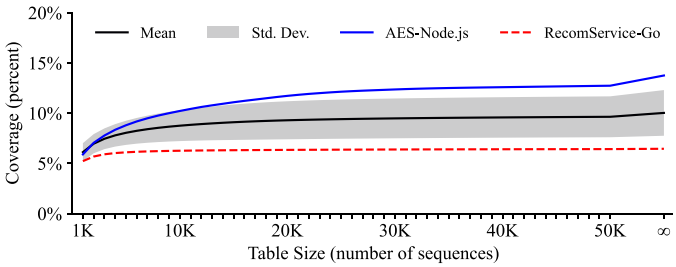


Fig. 3. Hardware memoization coverage as a function of memoization table size when memoizing only the ID-J sequences.

and calculate the total coverage of the top sequences for a range of table sizes. Fig. 3 illustrates the average ID-J sequence memoization coverage, showing the mean and standard deviation across all 26 workloads and highlighting the most and least sensitive applications (blue and red). A table with 10,000 entries already accommodates (on average across all workloads) 87% of the coverage of all repeating ID-J sequences. In the Recommendation Service of the Hotel Reservation Suite, the 10,000 most-frequently repeating ID-J sequences achieve 97% of the opportunity, indicating that the same sequences repeat throughout the entire execution of the application. Conversely, the 10,000 ID-J sequences that are most-frequently repeating in AES-Node.js cover 73% of the memoization opportunity, indicating that a fraction of the repeating ID-J sequences in AES-Node.js are not always hot, and suggesting that larger storage or better replacement policies are needed to capture longer-range repetition in this workload.

## VI. Conclusion

In this work, we analyzed the traces of 26 server applications to show the prevalence of cross-request similarity in server workloads, which makes them excellent candidates for instruction-level hardware memoization. We found that 93% of the dynamically executed instructions exactly repeat a previous execution, making them candidates for memoization. Although considerable work remains to develop mechanisms that leverage these observations, we identified sequences of repeating instructions starting with indirect jumps as a promising focus for a preliminary study. Our analysis showed that over 8.7% of instructions could be memoized using modest (on the order of 10,000 entry) hardware sequence tracking structures. We believe that our results make a strong case for further exploration of hardware memoization in future server processor designs.

## References

[1] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proc. Int. Symp. Comput. Archit.*, 1997, pp. 194–205.
[2] A. da Costa, F. M. G. Franca, and E. M. C. Filho, "The dynamic trace memoization reuse technique," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2000, pp. 92–99.
[3] D. Michie, ""Memo" functions and machine learning," *Nature*, vol. 218, pp. 19–22, 1968.
[4] S. E. Richardson, *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*. Mountain View, CA, USA: Sun Microsystems, 1992.
[5] Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 277–288.
[6] A. Suresh, E. Rohou, and A. Seznec, "Compile-time function memoization," in *Proc. Int. Conf. Compiler Construction*, 2017, pp. 45–54.
[7] J. Tuck, W. Ahn, J. Torrellas, and L. Ceze, "SoftSig: Software-exposed hardware signatures for code analysis and optimization," *IEEE Micro*, vol. 29, no. 1, pp. 84–95, Jan./Feb. 2009.
[8] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi, "ATM: Approximate task memoization in the runtime system," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1140–1150.
[9] G. Zhang and D. Sanchez, "Leveraging hardware caches for memoization," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 59–63, Jan.–Jun. 2018.
[10] Z. Liu, A. Yazdanbakhsh, D. K. Wang, H. Esmaeilzadeh, and N. S. Kim, "AxMemo: Hardware-compiler co-design for approximate code memoization," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Archit.*, 2019, pp. 685–697.
[11] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," in *Proc. Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2015, pp. 691–702.
[12] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima, "Design and evaluation of an auto-memoization processor," in *Proc. 25th IASTED Int. Multi-Conf. Parallel Distrib. Comput. Netw.*, 2007, pp. 245–250.
[13] M. Pilla, P. Navaux, A. da Costa, F. Franca, B. Childers, and M. Soffa, "The limits of speculative trace reuse on deeply pipelined processors," in *Proc. 15th Symp. Comput. Archit. High Perform. Comput.*, 2003, pp. 36–44.
[14] A. Amid et al., "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, Jul./Aug. 2020.
[15] EASE Lab., "vSwarm: A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks," 2022. Retrieved: Oct. 05, 2023. [Online]. Available: https://github.com/vhive-serverless/vSwarm