# Implementation and Utilization of Hardware Parallelism in Modern CPU Architectures

Justin Reed
Advisor: David Wonnacott

December 22, 2023

For at least the past three decades, nearly all computer architectures have made use of *pipelining*, a concept that has proven indispensable in improving the efficiency of computer processors, regardless of the application. Pipelining improves the throughput of an instruction sequence by running instructions in mutually independent stages, thus achieving what is called *instruction level parallelism* (ILP). This paper provides an overview of how ILP has been utilized to develop more efficient processors, reviewing a selection of papers influential to its use. To this end, the tradeoffs between out-of-order and in-order execution are considered. Finally, the most recent work being done to improve instruction level parallelism is discussed.

## Contents

## 1 Introduction

While efficiency in computer science is mostly considered at the algorithmic level, this efficiency is made irrelevant in the absence of a machine capable of executing it. Though this goes without saying, it is a reminder of the importance of considering hardware in the implementation of software. The same algorithm executed on two different devices could take either 1ms or 1000ms to complete, depending on the clock-cycles per instruction (CPI) of the CPU. Improving this is the task of hardware.

Much of the improvement to CPI has been accomplished through advances in the fields of physics and electrical engineering, but developments made in the designs of computer architecture have played an indispensable role as well. One of the first innovations was the switch from single-cycle to multi-cycle instructions [PH17b], shortening the clock-cycle by reducing the amount of hardware needed for each sub-instruction. Though the multi-cycle instruction is slightly slower than its single-cycle equivalent, it has the key advantage of making a pipelined architecture possible. Such an architecture begins executing new instructions before the previous ones have completed, taking advantage of the fact that each sub-instruction only needs a portion of the chip to execute. Multiple instructions are therefore able to execute concurrently, improving throughput by a factor close to the number of sub-instructions per multi-cycle instruction. Early processors used around five stages per instruction, and many modern processors now utilize up to fifteen or more pipeline stages, theoretically making possible an improvement in the instruction per clock-cycle (IPC) rate by a factor of 15, when compared to the single-cycle alternative. Even when considering the problems of register pressure, memory bottlenecks, data and control dependencies, and the various other system and hardware challenges limiting the IPC potential of the processor [PH17a], pipelining has still resulted in a substantial improvement to the speed of the CPU, making it essentially ubiquitous in modern processors. These processors are called *superscalar processors*, named as such for their efficient ability to utilize instruction level parallelism. The effectiveness of pipelining has further increased in the context of multithreaded architectures, the particulars of which are discussed in Section 3.

While any discussion of whether or not to use pipelining in an architecture is now a nonstarter, the particular ways of implementing pipelining have continued to evolve since its introduction. The trade-offs of these various pipeline strategies have been continuously debated. As the physical limitations of hardware change, architectures using one strategy may quickly displace architectures which use another, only for the old strategy to be ultimately reverted to when yet another breakthrough is made. One example of this is found in the use of *out-of-order* versus *in-order* execution, the details of which will be discussed in Section 4. Finally, the work being done to overcome memory latency will be discussed in Section 5.

## 2 Background

This section provides a brief overview of the technical concepts important to the papers covered in this literature review. A reader who is already familiar with the various topics concerning hardware parallelism will have no trouble skipping to the body of the review, which begins in Section 3. While this preliminary section serves as a sufficient summary of the fundamental topics of ILP, a much fuller understanding will be gained by reading through the relevant chapters of Hennessy and Patterson's *Computer Organization and Design* and *Computer Architecture: A Quantitative Approach.* [PH17b] [PH17a] Any fact not explicitly cited in this section is from one of these two textbooks.

As computer scientists, we are sometimes tempted to view computers as total abstractions, merely paper machines able to execute our theoretical algorithms, vaguely constrained by some idea of computation time. Indeed, if this were a true representation of computers, there would be nothing preventing us from adding unlimited cores with unlimited threads that make use of unlimited memory. Even if not all these resources would be used during the execution of a program, when physical constraints are ignored there would be no downside to such an implementation. Such a design, of course, is absurd and impossible, but even architectures that at first seem reasonable may still fall into this trap. Disregarding the economic and practical reasons why more is not necessarily better, the physical limitations of electric circuits prove to be a very real constraint.

One of the most thematic constraints is the amount of work able to be done on a computer in a single clock-cycle. Going back to our paper machine, we would theoretically like a single clock-cycle to be infinitesimal in length. But this is physically impossible, because computers are not, in fact, paper machines. They rely on electric signals which, though incredibly fast, are still limited by physics. A signal travelling from one part of the processor to another takes a nonzero amount of time, which is why we're not able to decrease the clock cycle without considering the physical impact it would have. The larger a CPU is, the longer the clock-cycle needs to be, decreasing the speed of the computer.

It is therefore difficult to increase the number of components on a chip without affecting the clock time. A possible compromise is to keep the clock cycle the same, but decrease the amount of work done during each cycle. This is the fundamental idea behind a multicycle implementation. In a multicycle architecture, the CPU is partitioned into sub-components that are each responsible for their own stage of execution. One instruction is now adapted by the compiler into several small instructions that accomplish a different stage of the original instruction. The way this instruction is divided can be changed based on the particular design, but it is now almost universal to divide the work done by the instruction memory, instruction decode, the arithmetic and logic unit (ALU), the data memory, and the register file into their own instructions. Over time, the clock cycle has been made even shorter by further dividing these stages into increasingly smaller steps.

As alluded to in the introduction, this design makes pipelining possible.

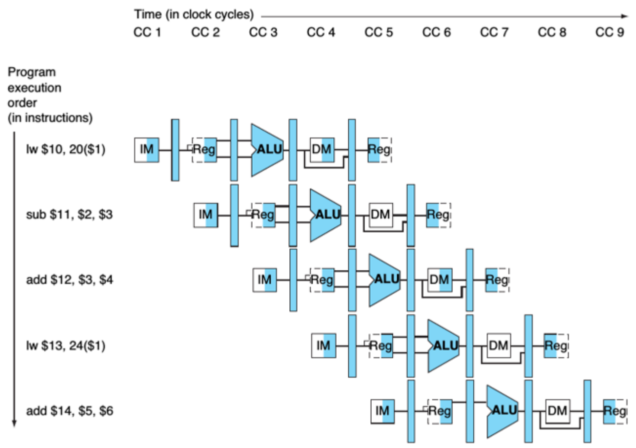

**FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions.** This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram.

Figure 1: Conceptual diagram of pipelining. [PH17b]

Multicycling by itself is insufficient to increase the speed of the CPU, but by facilitating pipelining its benefits become immense. A pipelined CPU can theoretically improve the number of instructions per clock cycle up to the number of instruction stages in the pipeline. This results from the processor being able to start a new instruction as soon as the first step of the previous instruction has completed. A conceptual visualization of this is provided in Figure 1. The CPU components used by the instructions at each clock cycle is depicted in the diagram. As the visuals make readily apparent, none of the instructions need to use the same section of the CPU at any clock cycle. In a naïve pipeline where it is always possible to do this, each CPU resource is utilized throughout the entire program, and the faster clock-cycle therefore becomes a real improvement in the speed of the processor.

In reality, many obstacles interfere with the ability of the processor to obtain this maximum level of instruction throughput (the rate at which instructions can be processed). These problems can be divided into two main categories: those resulting from data hazards and those resulting from control hazards. While the specifics of these problems vary based on the context, data hazards typically result when the pipelined processor attempts to use a value that would not have occurred at that part of the program if the processor had executed without pipelining. As for control hazards, they result when the pipelined processor attempts to execute a part of a program that would've been ignored in the absence of pipelining. Both these problems happen because the processor begins executing instructions before it knows if they should really be executed. The simplest solution to these types of problems is to introduce a stall, forcing the processor to act as if it wasn't pipelined when these problems might occur. This, of course, decreases the speed of the processor, so more creative solutions are desired, providing motivation for the papers covered in this literature review.

To resolve control hazards, one of the most effective tools is found in a technique called *branch prediction*. While branch prediction is not a focus of this literature review, knowledge of it is important to understanding various other hardware parallelism techniques, so a brief summary of it is provided here.

Branch prediction works by guessing which direction the program will take once it reaches a conditional branch. Branch prediction is necessary because the result of the branch instruction will not have finished by the time the instructions that follow it begin to execute. The processor therefore gets a head start by beginning to start on the instructions in what it believes to be the most likely branch to be taken. If the guess is correct, then the program proceeds as normal, and a stall will have been prevented. If the guess is incorrect, the processor "flushes" the incorrect instructions currently in execution. Since these instructions will not have finished by the time the branch prediction completed, they will not have made changes to the register file or to memory, so it is only necessary to disallow them from continuing their execution by disconnecting them from the next stage of the processor. As a result, it is as if these instructions were never issued, and it is functionally the same as if the processor had simply just stalled instead. Therefore, even if the processor blindly made a fifty-fifty guess, this approach would most likely still improve the efficiency of the processor, as there is almost no downside to an incorrect guess, other than the small amount of extra hardware necessary to prevent the instruction from entering the next stage of execution.

As it happens, however, there are many ways of improving the accuracy of the guess. While it might seem that making an accurate prediction would require more overhead than it justifies, most predictions are actually very easy to make. For example, once the program enters a loop, it is far more likely that it will continue to stay in the loop than to exit it. This type of prediction can be implemented using a two-bit branch predictor, which does not require much hardware. Such a predictor uses a history table, which is updated to predict a certain branch after it has already been taken twice in a row.

These are the most important concepts to understanding the papers covered in this literature review. While this preliminary background is not comprehensive, it should hopefully suffice as an introduction to the idea of hardware parallelism and the challenges that surround it.

## 3 Simultaneous Multithreading

One of the less artful ways to improve the efficiency of the CPU is to simply increase its number of components. While, as explained earlier, there is certainly a limit to the extent this can be done, it still remains a valid approach. As advances in electrical engineering allowed chip components to become more compact, it likewise became possible to fit more registers, memory, control units, etc. in a single CPU. But as the tasks of the computer increased in complexity, simply scaling up the resources of a single thread was no longer sufficient. Instead, a multithreaded approach was adopted, where several smaller CPU's were duplicated to create a multicore, multithreaded system. [PH17a]

For running several programs at the same time, the efficiency benefits of this approach were clear. Separate programs seldom interact with each other, and rarely share data spontaneously, so allowing them to run on separate cores requires very little control overhead at either the hardware, OS, or program level.

But even individual programs can take full advantage of these multithreaded resources. *Concurrency* is a tool used by programmers to implement *thread-level parallelism* (TLP) inside of individual programs. With concurrency, the order of parallel thread execution does not matter. So long as sequential program consistency is preserved at the broader program level, the processor can execute the separate threads in whatever order it chooses. Ensuring this is managed at the program level using thread locking mechanisms such as *monitors* or *semaphores*.

*Simultaneous multithreading* takes fuller advantage of this multithreaded structure. The techniques used by these architectures are thematically similar to how concurrency is utilized by programmers when designing multithreaded applications. While this approach doesn't directly address thread-level data and control hazards, it does serve to mitigate the inefficiencies caused by them.

ILP and TLP are essentially equivalent, the only difference is software parallelism can be viewed and manipulated by the programmer, while hardware parallelism cannot. This difference aside, at the execution level they can be considered interchangeably. There is no explicit rule requiring all the work done on a particular thread to be restricted to only one thread during execution. An instruction from one thread can execute on another, so long as it doesn't introduce any new data or control hazards.

So when TLP was made possible by the introduction of multiple cores, ways of further exploiting parallelism were considered. One of the most successful approaches was found in simultaneous multithreading (SMT), a concept put forward by Lo, Jack L. et al. in their 1997 paper "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading". [LEL+97] The architecture they proposed allows threads to share resources between them based on their availability. In such an architecture, ILP and TLP are equivalent, used by the CPU interchangeably as it runs. Figure 2 clearly illustrates the effectiveness of this approach, as many issue slots previously unused by the CPU because of data dependencies can now be used by other threads. Not only do the threads share pipeline resources, but cache memory is shared as well, allowing for improved synchronization between the threads.

The advantage of a SMT CPU is greatest for single-thread applications, as these programs are implicitly converted to multithreaded applications at execution time, allowing for thread-level parallelism where there previously wasn't any. But even for multithreaded applications, Jack L. et al. found that SMT CPU's maintained a clear advantage. The two and four core multiprocessors considered as baseline comparisons in the study had to choose between offer-

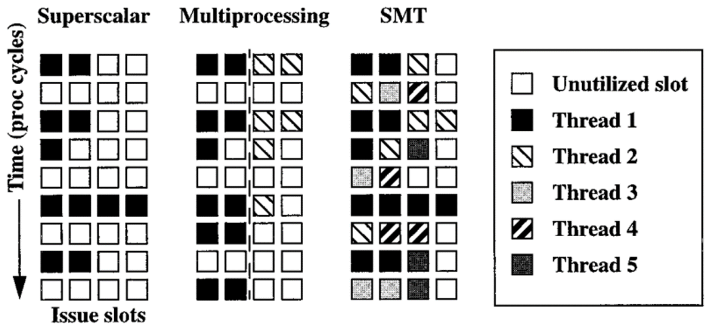

Figure 2: Equivalency of ILP and TLP in an SMT architecture. [LEL+97]

ing greater ILP or greater TLP, due to the limited space available on the chip. But since SMT is able to consider ILP and TLP interchangeably, the SMT CPU achieved high performance for both single-threaded and multi-threaded applications. On average, the performance increase from the SMT architecture was 52-percent greater than the highest performing multicore alternative. This performance increase was so decisive that nearly all modern superscalar general-purpose processors now make use of SMT. [SBGS22]

While clearly superior to prior designs, the authors of the paper acknowledged a few new challenges that SMT introduced, most notably problems regarding register pressure and memory latency. Ironically, both of these problems are primarily caused by the increased speed of the processor from the SMT design. *Register pressure* is a phenomenon induced by the processor resolving data dependencies through the renaming of registers. *Out-of-order execution*, which is discussed in Section 4, focuses on ways of resolving the problem of register pressure. [PH17a] Memory latency occurs for similar reasons. Since threads now share the same memory, there is more of a demand placed on the most quickly accessible tier of memory (the level 1 cache), so lower levels of memory may need to be used in compensation. Solutions to these memory related problems are discussed in Section 5.

## 4 Out-of-order and In-order Execution

While the effectiveness of simultaneous multithreading depends on the stalls caused by data and control hazards being inevitable to some degree, these stalls can be minimized. To resolve data hazards, various strategies have been developed to resolve data dependencies either statically at compile time or dynamically at execution time. In general, strategies using the former are labeled as using "in-order execution" (INO) while strategies using the latter using "out-of-order execution." (OOO) [PH17b]

In-order execution mitigates data dependencies by using the compiler to arrange compatible instructions into *very-long instruction words* (VLIW). This works by doubling the size of the instruction, hence the term "very-long." Two instructions are compatible if neither of them rely on the other as an operand. Note that only adjacent instructions can be combined in this way, as program order must be strictly maintained in this implementation, which is why it is called "in-order." If a section of program has true-dependencies incompatible for creating VLIWs, these instructions are left the same, and the CPU simply runs only one instruction at the times they are read. Of course, in the absence of additional CPU resources, it is impossible to run two instructions on the same thread at the same time. VLIWs are therefore facilitated by doubling the resources used for instruction execution.

While in-order execution has the advantage of requiring less overhead to ensure correct program order, it has considerably less flexibility in scheduling instructions to maximize efficiency. Out-of-order execution, which can dynamically schedule instructions at execution time, is less restricted in this way. Out-of-order execution secures most of its efficiency by permitting instructions to be inserted into the pipeline where stalls would have otherwise been necessary. From a human perspective, its very easy to rearrange instructions to this end, but for a computer this is by no means trivial. As such, the hardware shown in Figure 3 becomes necessary. The reservation station and the load and store buffers are the critical hardware components that make out-of-order execution possible. The reservation station resolves the false dependencies induced by write-after-write and read-after-right data hazards by serving as a virtual register file where the particular register used can be changed dynamically. Instead of being fixed into using a specific register by the compiled instructions, the register chosen in the reservation station for a particular instruction is now mostly arbitrary. Consistency is preserved by including a bitvector to keep track of the operands so that the instruction knows where to look when it is ready to execute. The reservation station thus abstracts the register into its critical path components.

Out-of-order execution allows for branch prediction to become more aggressive by adapting normal branch prediction into "speculation." In contrast to normal branch prediction, speculation will potentially finish several instructions before the result of the branch instruction is computed. This is made possible by the several temporary registers available for speculative instructions. Since these results are not committed until the branch prediction is confirmed, an incorrect prediction will not affect the accuracy of the program.

Instructions on a speculative branch can continue to be executed so as long as there is a supply of up-to-date operands, but no more productive work on the branch can be done if an exception is triggered. Beginning the exception protocol would actually be counterproductive. To examine why, consider this analogy: imagine a society where people filed a police report every time they lost their cellphone. While this would theoretically increase the response time of finding missing and stolen phones, the amount of resources necessary to facilitate this would not be worth it. To prevent the police from being overwhelmed, it would be necessary to establish a department dedicated just to processing stolen phone police reports. When considering that most phones would be

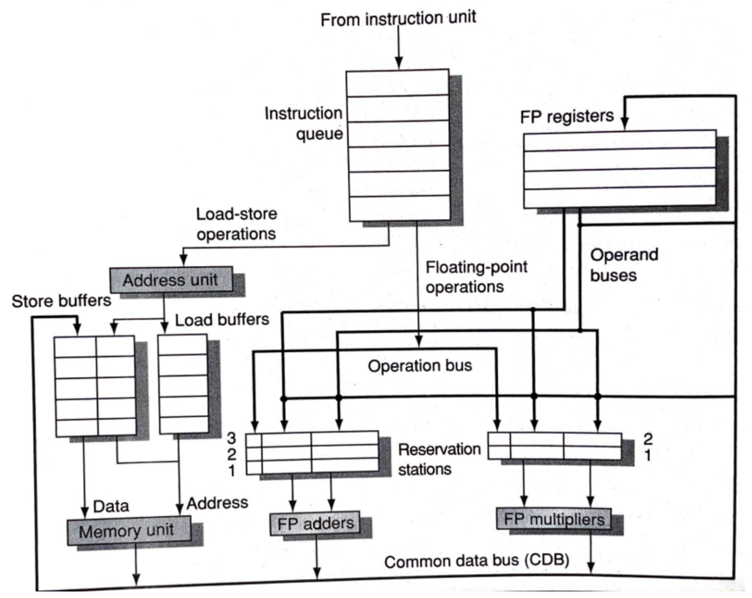

Figure 3: One possible implementation of out-of-order execution overhead. [PH17b]

found after just a minute or two of looking, the actual amount of people helped by this new policy would be insubstantial. In the same way, a thread that runs into an exception should wait and see if the branch prediction was correct before disrupting the entire processor with its exception protocol. This analogy becomes even stranger the further it's extended, but one can imagine the frustration of responsible people who seldom lose track of their phone (threads where exceptions are rare) at having to fund an entire department for the sake of a small group of paranoid people (threads with frequent exceptions). This analogy also demonstrates the typical trade-offs that must be considered when designing hardware. A new element added to a processor must necessarily come at the expense of resources for another, so the advantages of both designs must be carefully compared before favoring one over the other.

While in most applications out-of-order execution achieves higher throughput than in-order execution [HS99] [SW16], in the context of SMT CPU's, this advantage diminishes as the number of active threads increases. Consider the fact that the main advantage of OOO comes from its ability to reduce stalls induced by false dependencies through the use of register renaming. But since SMT allows for many of these stalls to be eliminated through thread sharing, OOO ends up contributing less to increased throughput. This result is examined in [HS99], where Hily and Seznec compare the speed of in-order and out-of-order processors for various numbers of threads and workload sizes. As Figure 4 indicates, in-order execution achieves an only 10 percent efficiency loss relative

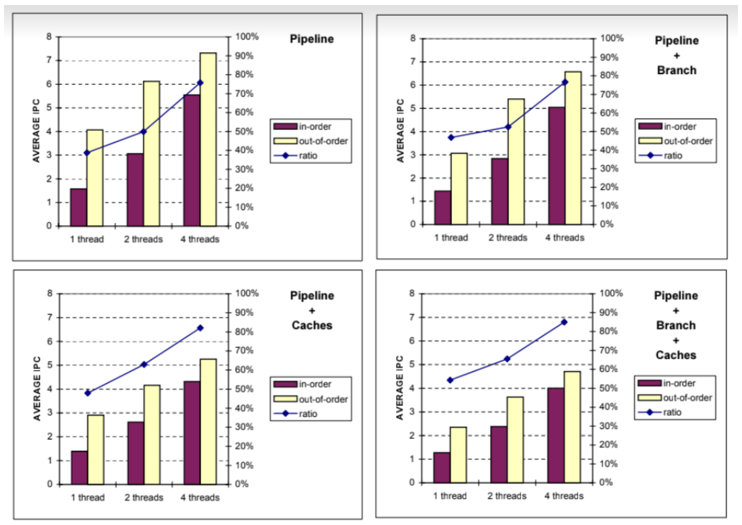

Figure 4: Advantage of OOO over INO decreases as the number of threads increases. [HS99]

to out-of-order in a CPU with four threads, branch prediction hardware, and increased memory.

This discovery may at first appear to have little practical use. Even if the advantage of OOO compared to INO is reduced in certain contexts, a small advantage is an advantage nonetheless; there would need to be stronger evidence to consider switching from OOO to INO processors. While this observation is true (at least for the large-majority of applications), it is only considering the highest level of the picture. Over the execution of an entire program, OOO is indeed faster than INO, but it is not necessarily more efficient at every section of the program. [SW16] To justify this claim, consider sections of a program where there are no control or data dependencies. In this case, register renaming facilitated by OOO would simply get in the way. Resources on the chip used to implement OOO would be better spent on INO in that case.

A CPU design able to maximize the advantages of both in-order and out-of-order execution therefore has the potential to improve efficiency. Indeed, when considering the results of [HS99], shifting to such a design may seem like an obvious choice, but as is often the case in CPU design, the whole is not always greater than the sum of its parts. In-order execution and out-of-order execution are only able to work by maximizing the use of divergent parts of the CPU: OOO by adding new registers and INO by adding new instruction-decode hardware and another ALU. A naïve hybrid architecture that scales both designs down by half and then simply merges them together will quickly encounter several new bottlenecks, leading to decreased efficiency. A more tactful approach is necessary.

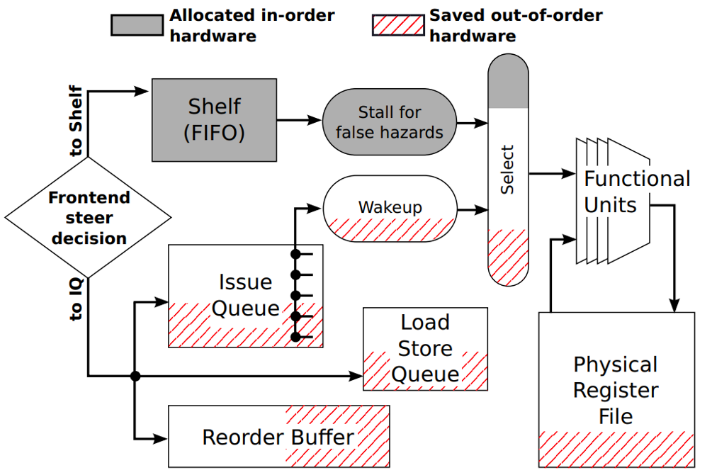

Figure 5: Sleiman and Wenish implementation of a hybrid INO OOO architecture. [SW16]

Such an approach is outlined by Sleiman and Wenisch in "Efficiently Scaling Out-of-Order Cores for Simultaneous Multithreading." [SW16] Acknowledging that OOO is more efficient in most cases, their architecture prioritizes preserving existing OOO resources. Instead, brief sections of instructions execute using INO hardware structures, mainly for the purpose of decreasing pressure on the OOO instruction queue. As instructions are issued, they are sent either to the out-of-order instruction queue or the in-order "shelf," as shown in Figure 5. Ideally, the instructions sent to shelf should have no dependencies, as these are instructions that would not have benefited in any way from out-of-order execution. Likewise, instructions sent to the OOO instruction queue should have dependencies that can be resolved by OOO execution.

This is very straightforward on paper, but coordinating OOO and INO requires clever architecture design. For example, consider that the instructions are separated at the issue stage. Unless control overhead is added, the processor could quickly become out-of-sync, with the in-order and out-of-order queues executing totally different stages of the program. Therefore, it is necessary to implement control hardware to keep execution organized. The most immediate problem is to ensure correct program order. This is done in their implementation by organizing sequences of OOO instructions followed by INO instructions into a single "run." To do this, the first OOO instruction to be dispatched to the instruction queue is marked with a bit. Since in-order instructions issue second in the run, they all must be younger than this instruction. Therefore, when the oldest OOO instruction is issued, the in-order instructions are permitted to run. Even with this condition, the instructions are still not fully coordinated. Since OOO renames registers during execution, the INO instructions need to keep track of the actual register being used. This is implemented using a tag system, where new tags are provided for the in-order instructions in addition to the physical register identifiers used by the existing OOO hardware. Several other edge cases are considered in their paper, the implementation details of which are complex. They are united by the overall problem of coordinating out-of-order and in-order implementation techniques, which, as has been shown in this section, are fundamentally different. Similar solutions to the ones just discussed are therefore used to provide this coordination.

Finally, the paper considers the problem of steering instructions to the "correct" instruction queue. Their hardware ensures that even if an instruction is not funnelled to its optimal queue, the consistency of the program will not be disrupted. Regardless, efficient steering means an efficient CPU, so prediction accuracy is still desired. Like the rest of their implementation, their steering mechanism is rather complicated, but its main functionality is provided by estimating the amount of time an instruction will take to execute. This time is calculated for both the INO and OOO queues, and the one that predicts less time is selected. Of course, this is somewhat hard to predict with precision. Instead of wasting process resources trying to do this, the total time of instruction execution is estimated by recursively adding a fixed score for each instruction in the data-path. This fixed score depends on the type of the instruction, a constant value known from the baseline architecture. Since loads can take variable time depending on cache-misses, it is assumed that all loads are L1 hits when making this estimate. This estimate gets updated as more accurate information becomes available, i.e. the instructions along the data-path finish execution.

Using this architecture, their experiments revealed an average efficiency gain of 11.5 percent. This figure demonstrates the merits of integrating in-order execution into a CPU architecture, even if on its own it is not as efficient as out-of-order execution.

## 5 Recent Bottlenecks: The Memory Wall

As alluded to in Sections 3 and 4, the advent of SMT and OOO execution have led to bottlenecks in CPU memory. Indeed, challenges with memory have the potential to be the most destructive to efficiency, as memory access is typically the slowest operation done by the CPU. Compared to simple arithmetic instructions that take between 3-12 cycles, memory access can require anywhere from 5 to 400 cycles in modern processors, depending on the level of memory that is accessed. [PH17a] [SBGS22] Even though lower-level memory access constitutes a low proportion of the instructions in most programs, [PH17b] the amount of time required for this class of instructions is substantial enough for it to present a serious obstacle in the design of efficient CPU's. This obstacle has become known as the "memory wall," [WM95] and is one of the main engineering problems faced by modern CPU designers.

As just alluded to, the first solution to this problem was to reorganize memory using a hierarchical structure. At the upper levels of the memory hierarchy, data takes less time to access, but in consequence there is less storage space available. As the memory hierarchy is descended, these facts reverse: storage space becomes greater at the expense of increased access time. The highest level of this memory remains the same: the register file, which, in following with the structure just outlined, has the fastest speed of access while being the smallest in size. The next level of memory includes the various caches, the amount of which can vary depending on the particular design. Main memory constitutes the lowest level of RAM, and is likewise the slowest to access.

If memory was statically organized in this way, allowing no (or minimal) movement between levels of memory, such a structure would present only a very minor improvement in the speed of memory access. As the portion of memory providing fast access must necessarily be small, then it follows that a very small section of memory would benefit from this modified design, potentially even at the expense of the access speed of the rest of memory. This is clearly undesirable, so permitting movement between levels of memory becomes indispensable. [WM95] Ideally, data should be moved to the highest cache prior to any memory access instruction being called. Failing this, the load or store instruction will encounter a 'cache-miss' and be forced to look for its desired address at lower levels of memory, further increasing access time for each subsequent level of memory traversed. Moving data that will soon need to be accessed up the memory hierarchy, a technique called *prefetching*, can reduce most of this latency. [SBGS22]

Shukla et al. take this approach a step further in their article "Register File Prefetching". [SBGS22] Building on previous memory prefetching designs, they not only allow prefetching to the highest cache, but also allow some memory access instructions to be bypassed altogether by prefetching into the register file. Their main argument is that while most data is successfully prefetched into L1 (the highest level of the cache), this still results in an inefficiency as the sheer quantity of data in L1 means that its slightly increased latency gets magnified. So while earlier papers focused on increasing the proportion of data prefetched from main memory into the cache, they argue that similar efficiency gains can be made by prefetching from the cache into the register file. Figure 6 illustrates this phenomenon.

At first, such an approach may seem more likely to do harm than good. As already noted, register pressure presents a significant obstacle to CPU throughput, so any unnecessary strain to the register file should be avoided. Indeed, in a statically compiled architecture where instructions for the most part execute in program order, their design would be impossible. Instead, they take advantage of the character of out-of-order execution, which, while improving overall throughput, has the potential to introduce significant latency to the execution of a single instruction. This latency results from instructions that have several dependencies, and therefore must linger in the reorder buffer for several clock cycles before their operands are ready. For such instructions, register prefetching becomes extremely useful, as the hardware can 'guess' the address they will need by the time they are ready. Once these memory access instructions are ready to execute, the CPU compares the calculated address to the predicted

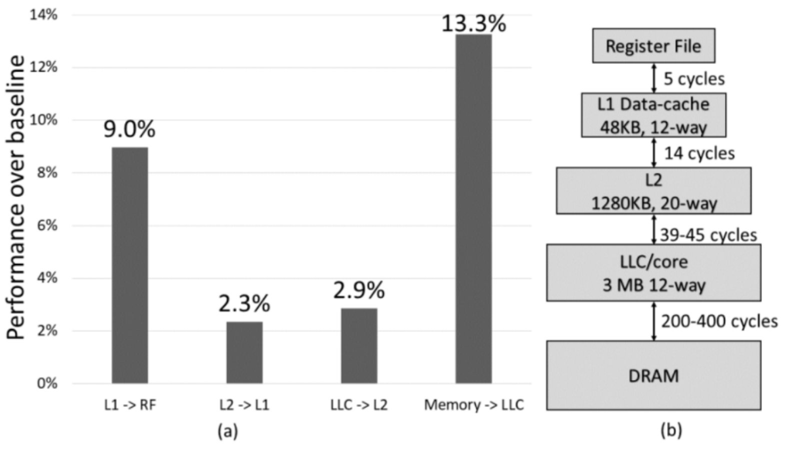

Figure 6: Potential performance improvement from successfully prefetching to next highest memory level for each level of memory [SBGS22]

address, bypassing the memory access stage if the address was correct. If it was incorrect, then the CPU proceeds with the memory access instruction as normal. Therefore, if the guess was correct, there is no increased register traffic, as the memory instruction would've placed the result there regardless. If it is incorrect, there is indeed increased register pressure, but since their implementation maintains a prediction accuracy of close to 90 percent, it is relatively minor.

In the ideal case where all memory is successfully prefetched into the register file, the benefits of their implementation are clear. There are, however, a few common cases Shukla et al. identified which prevent their CPU design from achieving this idealized level of efficiency. The CPU failing to prefetch the data in time is the most obvious one. Even though OOO allows a window to begin prefetching once the memory instruction is placed into the instruction queue, this window is still finite. The memory prefetch of course takes the same amount of time to access memory as a normal memory access instruction does. If the operands of the original memory instruction are ready before the prefetching has completed, the prefetching is useless. The impact of this on the CPU is neutral, however, as in such cases the prefetching can simply terminate and allow the memory instruction to proceed as normal. Indeed, in their experiments prefetching finished only 48 percent of the time. Of these, 90 percent fetched the correct data, resulting in register file prefetching being useful 43 percent of the time.

Another potential problem with register file prefetching is the increased memory traffic that may result. Considering that prefetching is not always successful, unnecessary cache traffic should be minimized. The first way this is done is by giving register prefetching a low priority of cache access, thus preventing it from interfering with in-flight memory accesses. Second, memory prefetching avoids further unnecessary traffic by only ever interacting with the highest level of the cache. If the address it is looking for is not in the L1, it stops and tries again later. While these two restrictions decrease the amount of data that can be successfully prefetched, they ensure that the consequence of a failed prefetch is at worst neutral. Following this principle is perhaps the most important rule of CPU design. All designs should at minimum have only a neutral impact on efficiency, and above all else should maintain program consistency. Shukla et al.'s conservative approach is as such. Even with these restrictions, their experimentation demonstrated a 3.1 percent performance gain over baseline.

## 6 Summary

"It is a strange fate that we should suffer so much fear and doubt over so small a thing." - Boromir, *The Fellowship of the Ring.* Boromir, of course, is referring to Sauron's ring of power, but these words could equally be applied to the CPU. The CPU is already so compactly organized, so efficient, and so unfathomably fast that it is hard to imagine making further improvements to it. Yet even incremental changes can have significant implications to the overall capabilities and efficiency of the computer. As cutting-edge computer workloads become more resource-hungry, the CPU must continue to improve.

To this end, this literature review covered some of the ways the CPU can increase its efficiency by making more intelligent use of instruction-level parallelism. Dissolving the boundary between hardware and software parallelism with SMT opened the door to further improvements to CPU efficiency. Out-of-order execution took full advantage of this to dynamically schedule CPU instructions. Its efficiency was improved by integrating a hybrid in-order architecture. Finally, bottlenecks to memory were addressed by prefetching data to higher levels of the cache, a strategy [SBGS22] expanded with register-file prefetching.

## 7 Future Work

While the work of [HS99] highlights an important result regarding the potentially overlooked disadvantages of out-of-order execution, the results of their experiment were obtained using a 1999 processor and workload. Replicating their paper and extending it for a modern processor or workload could provide insight into the current state of OOO versus INO execution.

In contrast, the work of [SBGS22] was done on a very recent processor, but its test suite only included single-threaded applications. Extending their tests to multithreaded applications would provide a compelling stress-test of their design. Considering the impact of multithreading on their design is therefore another potential area for future research.

Undertaking either of these proposals requires access to the designs used by

[HS99] or [SBGS22], the availability of which is not guaranteed. The resources of [HS99] may be easier to obtain as its design is older and thus less likely to be restricted.

## References

[HS99] S. Hily and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 64–67, 1999.

[LEL$^+$97] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, aug 1997.

[PH17a] David Patterson and John L Hennessy. *Computer Architecture: A Quantitative Approach.* Elsevier, Amsterdam, 2017.

[PH17b] David Patterson and John L Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Elsevier, Morgan Kaufmann, Amsterdam, Boston, 2017.

[SBGS22] Sudhanshu Shukla, Sumeet Bandishte, Jayesh Gaur, and Sreenivas Subramoney. Register file prefetching. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 410–423, New York, NY, USA, 2022. Association for Computing Machinery.

[SW16] Faissal M. Sleiman and Thomas F. Wenisch. Efficiently scaling out-of-order cores for simultaneous multithreading. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 431–443, 2016.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, mar 1995.