UNIVERSITY OF
CAMBRIDGE

In-core, hint-based, speculative multithreading

Márton Erdős

CHURCHILL COLLEGE

This thesis is submitted for the degree of Doctor of Philosophy.

July 2024

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted, or, is being concurrently submitted, for any degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Collaboration

This research was carried out in collaboration with others. Without their work, the results contained in this thesis would have been impossible to obtain. Throughout the thesis, I highlight work that was carried out by others, and I also give a brief summary here.

I claim the microarchitectures and microarchitectural optimisations presented as my own work. While these were discussed, I contributed the main ideas (except initial high-level plans, which are explicitly highlighted), and I was solely in charge of the implementation as well as the analysis and interpretation of results.

The architectural techniques described were developed in collaboration with others from Arm. While I contributed significant ideas, this area was not my responsibility.

The compiler, and the loop profiler (for workload analysis) were developed by others. While I contributed certain minor ideas, I do not claim ownership of these tools here. They are mentioned in this thesis on a high level only, in order to complete the overall story and give context.

I would like to thank my collaborators again:

- My academic supervisor, Timothy M Jones,
- My industry supervisors, Ali Zaidi and Giacomo Gabrielli from Arm, as well as the rest of the Arm team, working on the architecture, compilation, tooling and more,
- Utpal Bora and Akshay Bhosale, working on the final compiler, and
- Alexandra Chadwick and Yuxin Guo, working on loop profiling and program analysis.

Abstract

State-of-the-art high-performance processors rely on instruction-level parallelism (ILP) during sequential regions. This results in excellent performance in regions that exhibit large amounts of ILP. However, gains are limited elsewhere, due to strict upper bounds, superlinear scaling of costs, and sublinear returns. As the execution time of optimised regions decreases, Amdahl's law suggests that achieving good performance in the remaining regions is key for improving whole-program performance. On the other hand, maintaining good performance in high-ILP regions is just as important.

Thread-level speculation (TLS or SpMT, speculative multi-threading) has been identified as a possible solution in past academic work, but there has been a large disconnect between academia and industry in identifying design constraints and addressing practical adoption hurdles, which has led to underwhelming reception by industry. In particular, challenges remain in maintaining performance in high-ILP regions, minimising impact on the operating system and the architecture, and maintaining compatibility with high-performance microarchitectures.

I propose an in-core, hint-based, task-level speculation scheme to solve these challenges and efficiently speed up underperforming low-ILP program regions. Since the scheme does not introduce compatibility-breaking architectural changes, adding it to a modern high-performance out-of-order superscalar processor pipeline is feasible. I devise a set of crucial optimisations to overcome fundamental practical challenges, and describe a system for data forwarding in order to extend coverage to regions with frequent cross-task dependences. Finally, I identify the limitations and challenges that need to be solved to facilitate wide-scale deployment of the technology.

Acknowledgements

This thesis would not have been possible without many many amazing people around me. As well as during the last four years, I have a lot to thank for the journey that led me here. I will do my best to list some of you here.

First and foremost, I'd like to thank everyone who has been a mentor to me through the years.

I'd like to start with my parents, because their contributions could easily fill an entire book by themselves. They have given me unlimited love, care, attention, guidance, advice for decades, and – being excellent, dedicated Maths and Computer Science teachers/academics – they have taught me everything, directly or indirectly. Köszönök mindent!

I want to thank my academic supervisor, Timothy Jones, who has always gave me just the right combination of guidance and freedom to explore, and who was always available when needed. I could always work on what I wanted, and explore it the way I thought best, but always had someone to ask for a second opinion or ideas to move forward.

I would also like to say thanks to my first industry supervisor, Ali Zaidi. Ali has always had a strong bond with the project, and he has laid the groundwork for the explorations. We had numerous detailed discussions in first year, collaborating closely on obtaining results from compiler-microarchitecture codesign, and coming up with architectural refinements and additions. Even after leaving Arm, Ali remained passionate about the project and my achievements, and I'm happy we could catch up and chat multiple times.

My second industry supervisor, Giacomo Gabrielli also deserves a big thank you for being supportive and passionate about the project and my progress throughout the years, and for helping me develop and nurture connections with Arm.

Furthermore, my undergraduate director of studies, John Fawcett made sure my education was truly world-class, giving me very strong foundations in the wider field of computer systems and computer science in general. Apart from that, he has been an ongoing mentor and source of guidance and advice. Thank you for believing in me throughout, for all the advice and encouragement, and for never being too busy to talk (even though many people would have been in his position).

Let me wholeheartedly thank Matthew Ireland, who has been a mentor of his own will. He is an excellent teacher, and I really enjoyed working together, from undergraduate supervisions to mentoring the Churchill Compsci Talks together, to conducting interviews. I also truly value the advice he gave me and all the ways he helped me develop as a computer scientist, educator and person.

I want to also thank Sam Ainsworth, who has supervised me during my Master's project and undergraduate studies, and taught me a lot about asking the right questions, and making seemingly hard problems easy by finding the next step and just getting on with it.

My collaborators deserve another round of thank you. Alexandra Chadwick, for all the great discussions on architecture, research methods and more, Utpal Bora, for developing the compiler and working well together to produce results, as well as the newer members of the team, Akshay Bhosale and Yuxin Guo, who have quickly picked up work on compilers

and program analysis, and everyone at Arm (you know who you are) I've worked with. I'd also like to thank Arm for generously funding my PhD, and EPSRC (grant EP/W00576X/1) for funding the group.

Throughout the years, I've had many good teachers, and I am grateful to all of them. I would like to explicitly highlight Gyula Horváth and László Zsakó†, who prepared me for programming competitions, and taught me university-level material in a way that was approachable and exciting for my high school self. Also deserving a mention are my high school and elementary school Maths teachers, Anita Zsovár and Györgyi Gulyás, who constantly went above and beyond to provide me with engaging problems to solve, and not only tolerated my infinite energy, but helped me channel it into the right places.

I'd also like to thank my partner, Jiaqi Chen for being a source of joy (and sometimes good chaos) in my life, and for always being supportive whenever I needed it. I want to thank my brother, Gergő for being an excellent role model growing up, and for his almost-infinite calm, my grandparents with whom I have spent many afternoons during my school years and afterwards, and the rest of my family. Thank you to all of my friends who made this journey enjoyable, and to my teammates who held up my spirits (as well as the spirit of the game), even during the pandemic when we couldn't play together.

Thank you so much everyone, it's been a great time!

Contents

Chapter 1

Introduction

State-of-the-art processing leverages various forms of parallelism to obtain high performance [32]. In many cases, the problem exposes parallelism naturally. Custom and reprogrammable (FPGA-based) parallel accelerators are widely used in specific domains, GPUs can exploit regular and semi-regular parallelism in graphics and other embarrassingly parallel workloads, vector extensions in modern instruction sets allow for regular parallelism to be exploited in a wide range of simple loops, and finally now-widespread multi-core chips can exploit coarse-grained irregular (thread-level) parallelism from suitable programs.

Even for traditionally sequential problems in general-purpose computing, modern CPU microarchitectures rely on high levels of instruction-level parallelism (ILP) obtained from an increasingly long out-of-order window [21, 71, 90] to issue multiple instructions each clock cycle onto wide superscalar pipelines, in order to speed up execution. However, this approach has fundamental limitations, as expanded in Section 1.2.1. Growing the window size leads to increasing complexity, challenges and diminishing returns, and only limited ILP can be obtained from a finite window [44, 49, 87].

These factors lead to low performance and high levels of under-utilisation in the increasingly wide modern processor pipelines in low-ILP sequential regions of programs [93]. This thesis proposes to adapt and extend the idea of thread-level speculation (TLS, or SpMT, speculative multithreading) [17, 83] for the modern age to solve both of these increasingly important problems.

This chapter investigates sources of parallelism exploited in today's cutting edge processor designs. By considering their respective limitations, I characterise a type of parallelism that is not currently exploited by cutting-edge designs, and argue that exploiting it could result in significant whole-program performance gains. Then I derive the high-level idea for in-core, hint-based, speculative multithreading, a novel technique that aims to cover this type of parallelism without breaking compatibility with modern high-performance microarchitectures. Doing so paves the way for wide-scale deployment, which previous TLS schemes failed to achieve. The chapter concludes with the main hypothesis, and the outline of the rest of the dissertation.

1.1 The role of parallelism in modern processors

Processor performance has been improving exponentially from the first computers, all the way to today [32]. Historically, a large part of this increase has been driven by physical improvements in the manufacturing processes [53]. In particular, smaller feature sizes reduced propagation delays, as well as cost, area and power consumption per transistor, with every generation. These effects are widely known as Moore's law (which estimated the pace of progress remarkably well), and the reduction in power consumption (and closely coupled to this, heat generation) is known as Dennard scaling. While the rate of improvements (Moore's law) has merely slowed down in the last decade, Dennard scaling stopped around the turn of the century [16]. Anticipating and observing these effects, researchers and semiconductor companies have been looking at ways to continue delivering improvements. This thesis aims to further this effort.

Let us classify current performance improvements, as summarised in Figure 1.1. To derive this, we start from the the following basic formula for processor performance:

$$T = \frac{cycles}{f_{clock}} = \frac{instructions}{ipc \cdot f_{clock}} \tag{1.1}$$

where T is the execution time, f_{clock} is the clock frequency and ipc is the number of instructions executed per cycle on average.

Increasing clock frequency has a super-linear impact on power. This is explained by Hennessy and Patterson [32] in section 1.5 of their book. While at fixed voltage, frequency only increases power consumption linearly, in practice we also need to increase voltage to obtain higher clock frequencies (as it helps to pass the threshold voltage more quickly), which has a quadratic impact on dynamic power. Thus, significant increases in clock frequency are difficult, and require expensive cooling solutions.

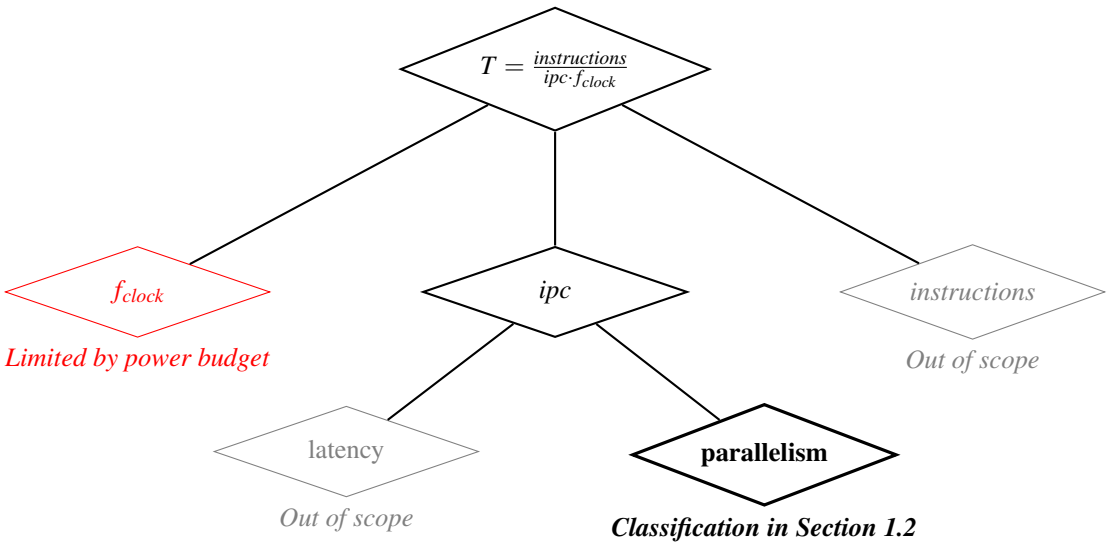


Figure 1.1: Main ways to improve performance of programs.

As shown in eq. (1.1), if we fix the frequency, there are two ways of reducing execution time: we can either reduce the number of (required) instructions, or increase the average number of them executed per cycle (IPC). The former is primarily targeted by algorithms research, however such optimisations are domain-specific, and the required expertise or development time required from the programmer often limit improvements in the real world.

This directs our focus to IPC. Since this is a throughput metric, we can increase it in two ways: either by reducing the latency (in cycles) of each individual instruction, or by overlapping the execution of multiple instructions in time, known as parallel execution. Latency improvements are targeted through numerous techniques, such as caching, prefetching, dedicated arithmetic units, offloading to a task-specific accelerators, improving memory latency, speeding up address translation. For these reasons I will primarily focus on parallelism.

Parallelism exists on multiple, largely independent levels (e.g. instruction-level, thread-level, data-level), which can – and often are – exploited simultaneously. However, each type of parallelism has its limitations. As explored in Section 1.2, such limitations restrict applicability to certain programs (or program regions), put a hard upper bound on the available parallelism, or yield diminishing trade-off curves (resulting in a soft limit on speedup).

1.1.1 The importance of coverage and performance on 'sequential' regions

We can view the impact of the above limitations from the angle of code *coverage*. That is, the fraction of execution time (or dynamic instructions) that meaningfully benefit from existing sources of parallelism. Crucially, as production systems can obtain extremely high performance on regions exhibiting certain types of parallelism, remaining regions have a disproportionately high impact on overall performance. Amdahl's Law [2] gives the formula for overall speedup S in the simple case, where a certain proportion p of execution time is sped up by a factor of s as:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

which tells us that the overall speedup S tends to $\frac{1}{1-p}$ as we increase the per-region speedup. Thus, unless p is close to 1, increasing s has diminishing impact on performance after impressive initial improvements. In the general case, where parallel speedup varies over time, the over-representation of under-performing regions in overall performance still holds.

For example, if we exploit instruction-level parallelism and obtain an IPC of 6 on the first 3 billion instructions of a program, but linear chains of dependences cause the IPC to drop to just 2 for the final billion instructions, then this last 25% of the program yields 50% of the run time. More ILP may not exist for the first portion for various reasons [87]. Furthermore, even if it would be possible to exploit more ILP, doing so requires a wider and deeper pipeline, and (likely) better branch predictors, prefetchers, larger caches and so on. These improvements are costly, and nowadays typically yield highly sub-linear speedups, as discussed in the next section (e.g. doubling the depth and width may increase performance by 30%, while it will increase area and power by more than 2×). This means that concentrating on the final (under-performing) quarter of the program may be more profitable. In our example, any technique that can achieve 30% improvement using any less than 2× area overhead is clearly

worthwhile. As we will see in Section 1.2.1, such under-performing (e.g. low-IPC) regions are common, even in cutting-edge high-performance designs today.

In summary, parallelism is one of the main ways to increase processor performance. While modern processors exploit multiple types of parallelism already, certain regions still exhibit subpar performance due to lack of profitable coverage by parallelisation. Furthermore, it is worthwhile investigating these regions, as their impact on run time is over-represented relative to their size.

1.2 Limits on parallelism

Let us investigate different forms of commonly exploited parallelism, specifically focusing on their shortcomings (coverage limitations). By the end of this section, this will lead us to identify thread-level speculation as a promising technique to cover regions of interest, which are largely untapped by other forms of parallelism (or only covered to a limited extent).

1.2.1 Instruction-level parallelism (ILP)

High-performance, out-of-order (OoO), superscalar cores exploit ILP by fetching and decoding instructions and renaming registers ahead in the instruction stream, tracking dependences within an out-of-order window, and attempting to issue multiple instructions per cycle. The window contains instructions that are subsequent in program order because instruction commit and frontend stages (instruction fetch and register rename in particular) process instructions in order (as doing so simplifies the implementation significantly[1]). Thus, without a paradigm-shift, parallelism is fundamentally limited by the window size, data-dependence chains and the predictability of branches. This is because instructions can only be issued from a limited window, and some of those instructions cannot be issued due to dependences, while others will only be in the window once a branch outcome is correctly determined (i.e. resolved after a misprediction). These factors restrict the amount of exposed ILP [44, 49, 87] and thus the ability of a large out-of-order core to mitigate the effects of performance bottlenecks such as memory latency and cache misses.

While improvements in individual components, such as prefetchers and branch predictors will keep pushing the boundaries, there are fundamental limits at play. In particular, branch prediction is paramount in limiting the impact of control hazards (such as branching) to hard-to-predict branches. While we can expect incremental improvements in the future, history-based branch predictors cannot tackle certain branches [94], even when given unlimited resources. Additionally, even to achieve such near-perfect prediction, significant area and power resources need to be used up, trading off frequency and power efficiency. Note that in order to produce a correct instruction stream, all unresolved branches in the window need to be predicted correctly, leading to exponentially decaying probability of correctness. For a given predictor accuracy p, the probability that an instruction past the n'th branch is correct can be approximmated as p^n, hence the expected number of correctly predicted branches in

[1] Fetch and decode resolve branches far more easily in order, register rename becomes very complicated if we do not know which registers will be written to, and commit needs to ensure in-order writeback to the memory system as well as handling any exceptions and interrupts correctly.

a row approximates:

$$\mathbf{E}(\text{Branches}_{\text{correct}}) = \sum_{n=0}^{\infty} np^n(1-p) = \frac{p}{1-p} \tag{1.2}$$

which translates to:

$$\mathbf{E}(\text{Instrs}_{\text{correct}}) = \frac{\mathbf{E}(\text{Branches}_{\text{correct}})}{f_{branch}} = \frac{p}{(1-p)\cdot f_{branch}} \tag{1.3}$$

instructions for a program with branch frequency f_{branch}. For example, a branch predictor with 99.5% accuracy will – on average – only correctly predict 995 instructions in a program where every fifth instruction is a branch (i.e. $f_{branch} = \frac{1}{5}$), even assuming an infinitely deep pipeline. This number is comparable to current cutting edge pipeline depths. Note that certain regions exhibit significantly worse prediction rates, highly reducing the effective depth (with a 95% prediction rate on the same program, we only get 95 instructions on average). Issuing instructions from both outcomes instead of prediction is in general even less feasible for the same reason: running past n branches would yield 2^n possible paths to execute, leading to only $\frac{1}{2^n}$ of the work being useful.

In practice, of course, the pipeline depth also has a hard upper bound, determined by the size of different structures (reorder buffers, issue queues, rename maps, physical register files), and control complexity. Pollack's rule [7] estimates the area and power requirements of a pipeline to correlate quadratically with the depth, making expected future improvements limited.

Out of the instructions in the issue window, the majority will likely not be ready to issue due to data-dependence chains. In particular, instructions dependent on a high-latency load will wait in the issue queue for a large number of cycles. Data value prediction has been considered for this purpose in the late 90s [47], and revisited in the last decade [63], but wide-scale industrial adoption has not happened so far, likely due to the high cost of mispredictions and low relative gains per correct prediction [63]. Close together instructions in program order often tend to be dependent (both for programming-related logical reasons and due to architectural register pressure), which may further limit ILP exposed.

All in all, while out-of-order processors are effective at extracting high amounts of ILP (often fully utilising a wide pipeline) in suitable regions (those with predictable branching, few long-latency loads and a high number of independent instruction chains), they struggle in regions that exhibit difficult-to-predict branches, many long-latency loads, or highly sequential dependency chains.

Apart from considering the absolute amount of ILP that exists in a region, it is also interesting to consider its granularity. As discussed before, ILP is only exploited within a window. Thus, extremely fine-grained parallelism is always exploited, but coarser parallelism may remain untapped due to window-size limitations.

In loops (or other sequences of quasi-independent regions), ILP may be exploited from within iterations or between iterations. The amount of useful ILP between iterations depends on the iteration size relative to the effective (run-time) window size. If multiple iterations fit into the window, then cross-iteration ILP exists. However, if iterations are large, and

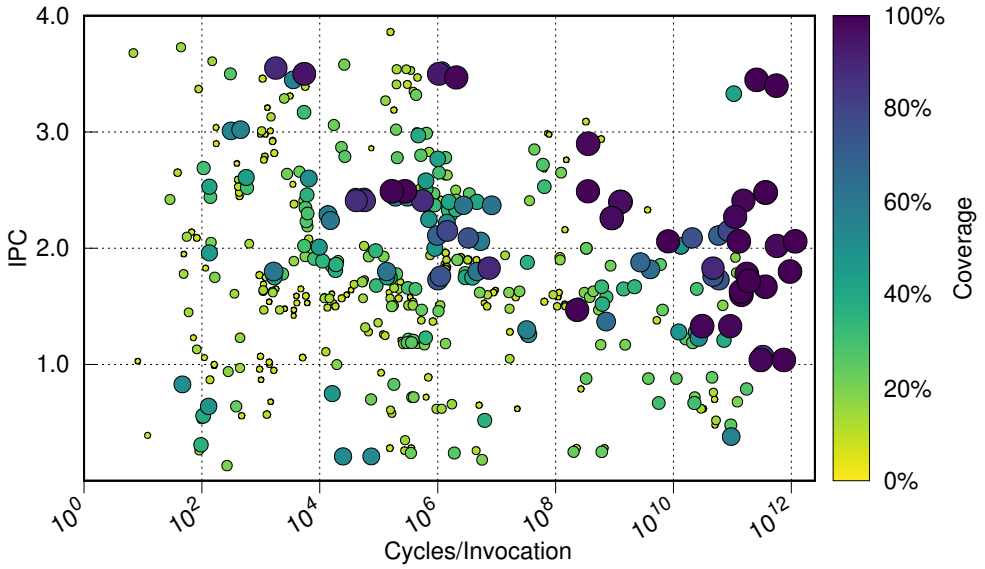


Figure 1.2: Loop IPC by loop duration in SPEC CPU2006 (C/C++ workloads), measured on an Intel® Xeon® W-2195 system (4-wide OoO superscalar, released in 2017).

Credits: Alexandra Chadwick and Yuxin Guo.

therefore the out-of-order window can only possibly interleave a short epilogue of the current iteration with a short prologue of the next iteration, then only minimal cross-iteration ILP is exposed.

Figure 1.2 shows average IPC values observed in loops in the SPEC CPU 2006 [33] benchmark suite. We can see that IPC values are highly variable. There are two important conclusions to draw from this. Firstly, given the majority of loops have IPC above 1, ILP is crucial for obtaining good performance in today's high-performance processors. It is important to retain the high IPC value for regions where it exists, otherwise performance will drop significantly. Secondly, however, for many regions, ILP alone cannot expose sufficient parallelism to fully utilise the pipeline. IPC values below 2 are very common, and some loops even exhibit IPC significantly below 1.

In this thesis I propose increasing utilisation – and thus performance – by simultaneously co-executing several parallel iterations in the same pipeline when ILP is insufficient, while retaining performance in regions exhibiting high ILP. Doing so has the potential to achieve meaningful overall improvements. Figure 1.3 shows a correlation between IPC and slowdowns obtained by doubling each program instruction, which demonstrates that the core is often under-utilised when ILP is low.

1.2.2 Thread-level parallelism (TLP)

Multicore systems and multithreaded processors exploit parallelism between explicitly defined threads. A thread is a logical unit of work, defined either by the programmer, a parallelising compiler [11], or a combination (e.g. programmer adds `#pragma` annotations and OpenMP uses them to choose loops and relax some constraints). Thread management is performed using a combination of software primitives that rely on atomic instructions, as

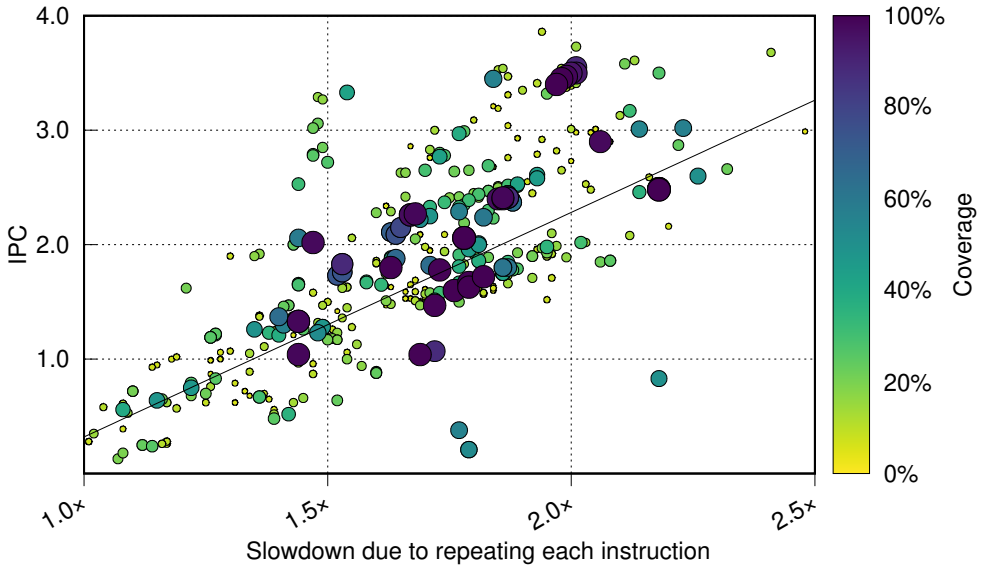


Figure 1.3: Measured slowdown per loop in SPEC CPU2006 (C/C++ workloads) after duplicating each program instruction, plotted against loop IPC. This simulates halving pipeline width and depth, and thus can be used to quantify general under-utilisation in the pipeline. There is a positive correlation ($r = 0.51$, $p = 10^{-32}$).

Credits: Alexandra Chadwick and Yuxin Guo.

well as system calls. Threads may be created and destructed on-demand, or the library may map software threads onto a pool of hardware workers using software scheduling. Care must be taken to handle data dependences correctly.

While thread-level parallelism can work well for some inherently parallel applications, and a lot of effort has gone into expanding coverage, a the inherent properties of TLP cause limitations [10, 93] that make a large number of regions not amenable to traditional thread-based parallelisation, as described below.

Firstly, manually parallelising programs takes significant skill and effort from the programmer, and parallel programming is well-known to be highly error-prone. Furthermore, it is far from guaranteed that optimal performance is achieved by a programmer. Parallelising compilers are a tempting solution to the above issues. Some good results have been achieved, but despite years of intensive research, a large number of workloads are not covered, which demonstrates the difficulty of the problem. The issue stems from the requirement to preserve correctness. This can be achieved by proving the lack of conflicts or by inserting synchronisation code conservatively. The former is difficult, the latter is expensive at run time, thus limiting coverage and speedups. Furthermore, code is often (incidentally) written in ways that hide parallelism, which means programmer expertise is still required to rewrite the algorithm. Additionally, at a fine-grained level, if complex control flow is involved, programmers may struggle to reason about dependences as well.

Thread management and synchronisation cost valuable cycles. Any system calls involved (for spawning/deallocating hardware threads or processes, blocking waits, inter-process communication, and signalling) require context switching into and out of the operating

system kernel. When threads run on different cores, accessing shared data (unless read-only) causes cache evictions (due to cache coherence), and therefore subsequent cache misses.

These costs are high for small threads or if frequent communication is required [3, 8, 10], but the overheads start to be amortised away if the parallelism is coarse enough (that is, communication is infrequent and threads are long-lived). Tiling (grouping multiple iterations together) may help, but regions with too few iterations or frequent-enough communication are left uncovered, and medium-grained parallelism is left unexploited in any case.

In conclusion, exploiting TLP requires programmer expertise, and it only applies to coarse-grained parallelism.

1.2.3 Vector processing for data-level parallelism (DLP)

Short-vector extensions – in general-purpose ISAs – exploit data-level parallelism by operating on data vectors instead of scalars. Machine code explicitly uses vector instructions. As with multithreading, these can either come from the programmer (intrinsics), or a vectorising compiler.

The former is sometimes performed in performance-critical code, the latter is routinely caried out by mainstream compilers (e.g. gcc, LLVM) on high optimisation levels. Still, coverage is largely limited to 'embarrassingly parallel' loops, because – even more so than for TLP – conflicts must be largely absent, and their absence needs to be proven. Code versioning may help, but has its own overheads, including extra instructions, data-dependent hard-to-predict branches, and increased instruction-cache pressure.

Furthermore, efficient support for irregular control flow is extremely limited, even in the newest, most complicated vector extensions. Masking becomes inefficient when control flow diverges [81], and many common control-flow patterns – such as non-trivial inner loops – are not typically supported. Auto-vectorisation is also subject to the limitations of alias analysis to avoid illegal dependences, which is widely known to be challenging, and uncomputable in the general case.

Thus DLP parallelism is only applicable to simple regular loops, usually inner loops. Although there is ongoing research efforts to massively expand the scope of outer loop vectorisation [15, 81], fundamental limitations around control flow divergence [81] remain open challenges for now. Often, vectorisable inner loops are nested inside more complex outer loops, leaving possible additional parallelism (orthogonal to DLP) unexploited.

1.2.4 Accelerators and other specialised processors

Frequently performed parallel tasks may be sped up by hardware accelerators. These can be full-custom ASIC, reprogrammable FPGA, or programmable using a domain-specific language. GPUs are also widely used for exploiting single instruction multiple thread (SIMT) parallelism in code.

The main limitation of accelerators is coverage (area constraints dictate that only very frequent tasks are covered). Furthermore, both accelerators and GPUs suffer from high setup costs (code and data must be offloaded from the CPU), and programmer effort required. Setup costs are only amortised for tasks that are sufficiently long running, and effort will

only be spent on very hot code. Further to this, GPUs also need code to be fairly regular, as diverging control flow leads to poor performance. These factors limit coverage.

I argue that accelerators and GPUs are incidental to CPU performance. Many workloads exhibit specific types of – typically fine-grained – parallelism, which can be exploited exceptionally well on custom accelerators. However, the necessity to run general-purpose workloads means CPU performance remains highly important. Therefore, instead of competing with accelerators on accelerator-friendly kernels, we should consider how to improve CPU performance across the range of workloads that would normally run on a CPU.

1.3 Unexploited parallelism

Let us summarise and visualise the above points using some simple experiments. This section aims to use the limitations of each type of parallelism to characterise loops that likely have high amounts of unexploited parallelism. Although additional parallelism is not limited to these loops, a scheme targeting it should aim to cover these loops. I use the characterisation to establish some design goals for the parallelisation schemes described in later sections.

1.3.1 Characterisation

Figure 1.4 shows the distribution of loops in SPEC CPU 2006 (C/C++ workloads) based on cycles taken per invocation and per iteration. Additionally, it shows the loops whose iterations could potentially be parallelised by traditional TLP and ILP techniques – and those that fundamentally do not have significant parallelism – based on these parameters.

Loops near the $x = y$ axis have very few (often just one[2]) iterations, and therefore they do not contain significant parallelism between iterations.

As discussed in Section 1.2.1, while ILP can extract intra-iteration parallelism for most loops, it can only extract cross-iteration parallelism if multiple loop iterations fit fully into the effective out-of-order window. Here, we estimate this to be loops spanning at most hundreds of cycles per iteration. For loops with low IPC or hard-to-predict branches, the effective window size can be significantly smaller than this, making cross-iteration ILP ineffective.

To amortise the setup costs of traditional multicore TLP, loop duration generally needs to be on the order of 10^5 cycles or longer [3, 8]. As described in Section 1.2.2, other factors – including synchronisation cost and programmer expertise – still make most of these loops not amenable to TLP, so this is a generous over-estimation of coverage. TLP also requires coarse iterations. While this can sometimes be ensured by increasing the chunk size (thereby packing multiple consecutive iterations into each thread), this reduces parallelism and increases synchronisation costs for loops with frequent cross-iteration dependences. Additionally, due to the minimum profitable iteration size, even if TLP exposes some cross-iteration parallelism, additional, orthogonal parallelism remains between the iterations within each chunk.

As shown in Figure 1.4, an uncovered area remains in the middle. This thesis attempts to devise techniques for targeting cross-iteration parallelism primarily for these loops, but

[2]This can occur for example in hash tables, where the program only iterates if and when a clash occurs, or other logically similar patterns.

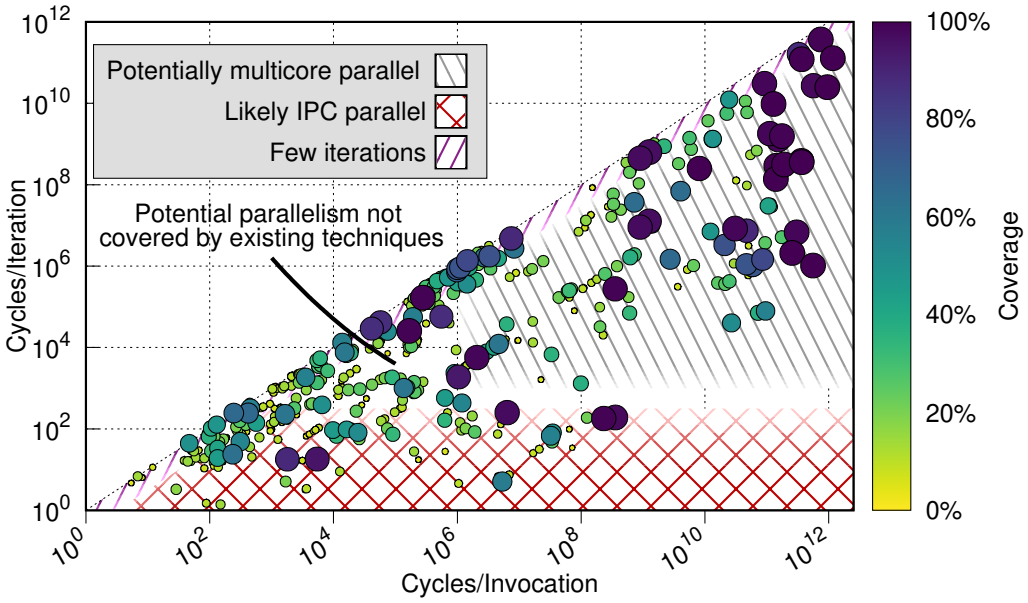


Figure 1.4: Loops in SPEC CPU 2006 (C/C++) by invocation and iteration duration, showing potential for ILP and TLP based on these factors. The boundaries are blurred, and other factors often make loops ineligible. There is a region with unexploited parallelism.

Credits: Alexandra Chadwick and Yuxin Guo.

it is expected that many loops outside of this triangle will still see benefits, due to the aforementioned additional limitations of the other techniques.

Data-level paralellism (vectorisation), and accelerators are not shown in this diagram. This is because – as described previously – their requirements are based on other factors. Accelerators and GPUs require an explicit invocation and domain-specific code, while DLP and GPUs require regular loop structure. Additionally, DLP is often fine-grained and applies to one loop level only, leaving orthogonal parallelism, while accelerators and GPUs suffer from similar setup cost (system calls, data movement) limitations as TLP.

1.3.2 Requirements

Based on the characterisation in the previous sections, to exploit the additional parallelism present in applications we should target irregular parallelism coarser than ILP but finer than TLP. This means exploiting parallelism between iterations with hundreds to thousands of instructions, including in loops executing in as few as 10^4 cycles. To achieve this, a scheme needs to:

- handle irregular control flow,
- be able to parallelise/reorder instructions thousands of dynamic instructions apart,
- have low per-iteration costs, which amortise within hundreds of cycles,
- have low setup costs, amortised within 10^4 cycles,
- require minimal or no involvement from the programmer, and
- place a low burden of proof on the compiler.

This points to a technique that combines features from both ILP and TLP. The technique presented in the next section and later in the thesis achieves this, and it is also cleanly composable with vectorisation, thus also able to extract existing, already-exploited DLP.

1.4 Approach

As discussed in Sections 1.2.1 and 1.2.2 (respectively), ILP and TLP both struggle to satisfy the above requirements. I argue that thread-level speculation (TLS) combines some of the advantages of both, giving a promising path forward. However, existing TLS schemes do not resolve all the limitations, and it they add significant deployment barriers. Finally, this section introduces in-core, hint-based, speculative multithreading as a possible solution to these issues. The rest of this thesis explores this technique.

1.4.1 Thread-level speculation

TLS [17, 83] aims to combine the benefits of ILP and TLP. In its basic form, it utilises threads to avoid the limitations of ILP parallelism, but uses additional mechanisms to make them abortable in order to relax the proof of correctness requirement and allow for correctness checking to be delayed until *after* thread execution. Chapter 2 expands the high-level description given here.

To obtain multiple speculative threads, observe that we can often make accurate high-level control-flow predictions, even if we cannot predict all intermediary branches precisely. A good example is a loop: we know that the next iteration will very likely execute at some point in the future, even if internal branches within the current iteration are difficult to predict. Predictions can be made either statically, at compile time, or dynamically, at run time.

To handle any control-flow mispredictions or incorrect data-flow (e.g. invalid operation ordering) between threads, an abort mechanism is needed. Some threads – often all except the oldest one – are designated as speculative and can be squashed. For speculative threads to permanently modify (architectural) program state they need to successfully *commit* first, which is conditional on some correctness checks. Aborting (often called squashing) is often facilitated using an undo log or some form of multi-versioned cache/memory structures. Control-flow mispredictions can be identified using bounds checking (e.g. of loops), either at run-time or using static annotations (e.g. on loop exits). Data-flow mispredictions can be identified by tracking read and write addresses (and possibly their order) and checking either on each operation (eager conflict checking) or at commit time (lazy conflict checking).

However, as highlighted in Chapter 2, the high costs seen in traditional multithreading remain mostly unaddressed in prior TLS schemes. This is because the use of architecturally visible threads and multicore setups require OS involvement and/or highly disruptive microarchitectural changes. Furthermore, deployment of such schemes has proved to be very challenging in practice [17, 93] due to their impact on modern architectures (including the memory model), high-performance microarchitectures, and – crucially – compilers and operating systems.

1.4.2 In-core, hint-based, speculative multi-threading

In this thesis, I propose in-core, hint-based, speculative multi-threading, which tackles these problems by moving speculation inside a single core and into the microarchitecture. As explored in later chapters, doing so cuts run-time costs and highly simplifies deployment.

To summarise the main differences from traditional TLS, I use microarchitectural-only *threadlets*, instead of architecturally-exposed *threads*, in order to perform speculation in a transparent way. I use simple architectural hint instructions to ease the microarchitecture's job and build on the strengths of the compiler's static analysis. The use of hints maintains transparency, thus keeping the architectural, OS and compiler footprint small. No programmer input is required, although annotations may be provided to improve hint insertion.

By doing this, we move slightly closer to ILP in order to eliminate TLP's thread management and communication costs and TLS's deployment complexity. The architectural hints are used to extend the reach of the processor's out-of-order window by splitting it into multiple disjoint sub-windows. Allowing instructions to commit to speculative state allows for pipeline resources to be recycled and threadlet state to be compressed (as only memory read and write sets and written values need to be stored, instead of all instructions), which allows speculation further into the future. High-level control-flow predictions help to increase accuracy, enabling the jumping ahead to find potential parallelism.

Even though restricting to a single core limits the maximum gains possible compared to traditional TLS, I argue that this is a worthwhile trade-off so long as it unlocks practical, meaningful speedups on top of state-of-the-art modern CPUs.

1.5 Hypothesis and thesis outline

> **Hypothesis:** *In-core, hint-based, speculative multithreading can be added to a realistic high-performance out-of-order superscalar processor pipeline, with minimal impact on the instruction set architecture, and without disturbing existing microarchitectural techniques and optimisations. Thus, adding this new capability produces meaningful, practical performance gains over a state-of-the-art baseline.*

The main chapters of this thesis set out to prove the viability of in-core, hint-based, speculative multithreading in high-performance production systems. Starting with an architectural proposal and a working microarchitecture design (Chapter 3), I find that several novel techniques need to be added in order to obtain meaningful performance gains (Chapter 4). Finally, I investigate the possibility of expanding coverage to more loops by handling more complex dependences between parallel epochs (Chapter 5). Chapter 6 draws conclusions, revisits this hypothesis, summarises future work, and discusses the outlook of the technique into the future.

Chapter 2

Related work

This chapter presents a survey of the wealth of work previously published about TLS. While there have been numerous studies, including some showing promising results, mainstream industry adoption [12, 56] has been lacking so far. The focus of this section is to understand why this is, and how we could move on as a field.

The area of TLS attracted a large amount of interest in the nineties and early two thousands. As detailed later in this chapter, papers typically focused on exploiting fine-grained parallelism between tasks of tens to hundreds of instructions. In the meantime, industry developed a different solution to the same problem. Deep and wide out-of-order processors now routinely exploit parallelism present at this granularity [30, 62], and advanced front-end pipelines can keep the back end full enough to facilitate this [36]. Thus, in order to find orthogonal parallelism and improve system performance today, a coarser approach is required.

Summaries and limit studies are valuable, but they do not provide specific, detailed proposals. As explained in the main chapters, measuring and understanding low-level microarchitectural phenomena is crucial for an accurate characterisation of performance.

Similarly – to the best of my knowledge – all prior TLS proposals have been evaluated using trace-based simulation, or other high-level simulation techniques only. These methodologies do not test the feasibility of implementation, and they ignore important factors affecting performance, such as instructions processed in the shadow of a mispredicted branch.

Finally, several proposals have a significant impact on the architecture (e.g. memory model), code generation, or both, which is typically treated as a second-order concern. While this allows innovative and effective designs, I argue that adoptability needs to be a primary objective for a scheme to be widely deployable.

In summary, I argue that there is a significant gap in terms of research focusing on achieving realistic but tangible gains in a way that is directly applicable to modern high-performance (micro)architectures. Filling this gap could provide the much needed performance gains [93] in the field, in the foreseeable future.

2.1 Summaries and limit studies

There are several survey papers about TLS techniques. Perhaps most notably Josep Torrellas composed a summary [83] concentrating on types of parallelism, as well as the common features and categorisation of TLS approaches. Furthermore, Estebanez et al. wrote a detailed review [17] of approaches and techniques, as well as other limit studies. The latter survey paper was my main resource for composing this literature review.

The results presented in TLS limit studies vary widely, as hidden assumptions have significant effects on the results. Oplinger et al. [57] showed encouraging results using a limited number of threads and realistic prediction (i.e. last-value and stride prediction). However, they claim that speculating on procedure boundaries as well as loops is paramount to unlocking this speedup. This view was later challenged by the authors of STAMPede [75, 76]. After their compiler eliminated obvious data dependences and performed explicit synchronisation, loop speedups increased significantly (see Section 2.3.1). Clearly, it is difficult to model the space of possible compiler transformations in a limit study, and manual (algorithmic) optimisations are even harder to capture.

Similarly – as summarised by Estebanez et al. [17] – Prabhu and Olukotun [64] asserted that significant parallelism exists in SPEC CPU 2000, the exploitability of which was then challenged by Kejariwal et al. [40], who also claimed only 1% possible improvement on SPEC CPU2006 [39]. Later, both of these new results were questioned by multiple papers [35, 60]: these studies only considered innermost loops, where parallelism is limited.

In summary, untapped parallelism can likely be exposed by thread-level speculation, although profitably exploiting it may be difficult, and even small changes or limitations can destroy all of these gains. Thus, it is important to focus on the right targets. Recently, Loopapalooza [93] categorised loops by types of dependences and predicted good speedups. I discuss Loopapalooza in more detail in Section 2.8.2, as its characterisation had a significant effect on the structure of this work.

2.2 Multiscalar

Although speculation was already explored before [41], the first fully fledged TLS implementation proposal was given by a 1995 publication by the Multiscalar project [20, 74, 86]. This is a hugely influential, foundational piece of work in the field. Broadly, Multiscalar consists of two main parts: the *Multiscalar paradigm*, which defines a new model of computation, and the *Multiscalar processor*, which is a specific implementation of the paradigm, which I discuss below.

2.2.1 Multiscalar paradigm

The Multiscalar paradigm [20, 74] is the first to describe TLS. It redefines program execution as a walk of the control-flow graph (CFG). To speed up this walk, the authors introduce the concept of *tasks*. We can define a task statically, as a connected subgraph of the CFG, or dynamically, as a contiguous section of the dynamic instruction trace (this is later referred to as an *epoch*). Tasks may speed up the CFG walk by allowing the processor to take larger,

task-sized steps (in addition to the usual single-instruction steps). Tasks can also be executed in parallel, as long as sequential semantics are preserved by honouring inter-task dependences through both registers and memory.

The authors use compiler annotations to delineate tasks, but also acknowledge that this is a design choice. The compiler uses heuristics to limit task size, inter-task dependences, and the number of successors. There are also some additional annotations to define register control flow: the *create mask* defines the conservative write set – that is, the set of registers each task *may* write to – and the *release point* for each register marked in the create mask, which defines the point when the register has (provably) reached its final value. The final value may be reached either straight after the last instruction that writes to the register, or after executing the last conditional branch that could have lead to a code path containing more writes (e.g. after exiting a loop that repeatedly updated the register, or skipping a conditional update).

A task starts with all the registers marked as busy, but each busy flag is removed as soon as the corresponding (input) register is received from a predecessor. This could happen either at task launch for registers that are ready in the predecessor (i.e. if it has already received the register from its predecessor), or when the predecessor forwards a value later. Forwarding happens when a (ready) register is released, or when a register not in the write set is received from the predecessor.

Using this system, tasks can precisely synchronise register values: by delaying the execution of instructions that read registers that are not ready yet, all register dependences are honoured, since the write set is conservative (an over-estimate). Other possibilities, such as eager (speculative) forwarding, or data speculation are also considered.

For memory dependences, the authors notice that accurate static analysis is difficult, therefore using conservative write set annotations would reduce parallelism too much due to too many dependence edges. Instead, the microarchitecture handles memory dependences. It speculates that no dependences exist, buffers speculative memory updates, monitors possible conflicting addresses, and *squashes* (discards and restarts) successor tasks if a dependence violation is noticed.

2.2.2 Multiscalar processor

Multiscalar targets very fine-grained parallelism (on average 13 ± 5 instructions for integer benchmarks and 68 ± 28 for floating point programs from the SPEC CPU '95 suite). Doing so keeps things easier, as small tasks exhibit fewer register dependences and memory conflicts, they have fewer successors (which multiscalar needs to predict), as well as smaller speculative state. However, today's high-end processors already efficiently exploit parallelism at this granularity[1] via out-of-order execution (ILP), thanks to – among other improvements – a large-enough window to fully contain many such small tasks at once, wider pipelines, and better branch predictors. Furthermore, as discussed in Section 3.12, the need to fill up and drain the pipeline so frequently would introduce a large overhead on a modern large core.

[1] We measured a geometric mean IPC value of 2.0 over SPEC CPU95 on an Intel® Xeon® W-2195 processor (4-wide OoO superscalar, released in 2017), which is higher than Multiscalar's simulated 1.9 IPC with 4 processing units over SPEC CPU92 (and visually similar IPC over CPU95).

To exploit this parallelism, the Multiscalar processor features multiple processing units (PUs), organised in a ring [86]. Each unit acts mostly as an individual processor, with its own pipeline, architectural state (PC and registers) and microarchitectural structures (such as reorder buffer, issue queues, load-store queue, etc.). However, neighbouring units can efficiently communicate using a forwarding ring for register values and release signals, and all PUs collaborate to execute the same program. Tasks are mapped to the PUs in order, along the ring. The head and tail units are tracked using pointers, with the head executing the oldest (non-speculative) task, while the rest are executing younger (speculative) tasks. The memory system features a new address resolution buffer (ARB) on the DRAM side of the memory interconnect, which buffers all stores from PUs running speculatively. Load and store addresses are also tracked and compared in this central ARB so that conflicts can be identified, and offending tasks squashed. Multiscalar describes the forwarding and squashing logic in detail, but the internals of the ARB are left more vague.

The largely self-contained nature of the PUs was seen as a strength of Multiscalar at the time. As spelled out by Rotenberg et al. [68] for their trace processor, the hierarchical subdivision reduces scaling complexity. As Pollack's rule formulates, processor complexity (area) scales quadratically with performance. However, making N mostly-independent PUs only results in an approximately linear increase in area, and could – in an ideal case, if fully utilised – increase performance N-fold, thus effectively saving a factor of N on area and design complexity. Unfortunately, this observation ignores three important factors: utilisation, conflict rates and communication latency. Mapping tasks in a way that constantly utilises all PUs is tricky, especially with larger tasks and if the conflict rate is to be kept low. Furthermore, fully-accurate conflict avoidance would further reduce utilisation due to added (conservative) delays, and the resulting logical complexity is identical to the requirements of a single, wide pipeline with a deep speculative window. Additionally, if larger cores are joined together in a similar way to Multiscalar, the communication latencies will be similar to chip multi-processors (CMPs), therefore the utility of a single Multiscalar core over a CMP is not clear. Hence, modern mainstream processors evolved to feature fully-fledged multi-core support, utilising powerful, deep and wide out-of-order pipelines

Realising this, following Multiscalar and other early papers [68], the field of hardware-supported TLS evolved in two main directions. Schemes based on CMPs (or multicore), focussing on coarse-grained parallelism, and schemes based on simultaneous multi-threading (SMT) inside a single pipeline, focussing on fine-grained parallelism, as explored in the following two sections.

2.3 Multicore TLS

Krishnan and Torrellas [43] proposed to scale the Multiscalar paradigm to multiple cores in a core cluster, thus opening a rich field of CMP-based TLS. Proposals at this time were nominally multi-core, but due to the use small, tightly coupled cores, they assume very low cross-core communication latencies by today's standards (e.g. 1-3 cycles). STAMPede [75] was the first to propose scaling to a full traditional CMP. The SWARM project [37, 92] proposed scaling to a chip with a very large number of cores, thus making more of an accelerator-like architecture. I detail these two prominent schemes (STAMPede and SWARM)

below, as they are fully-fledged implementations, building on other advances, and they show two distinct points on the cutting edge of CMP-based TLS.

2.3.1 STAMPede

STAMPede [75–77] is a comprehensive TLS scheme, which claims to scale well, from a single SMT core, through a chip multi-processor (CMP) all the way to a multi-chip system. However, since the evaluation only tries the CMP case – and this seems to be the main focus of the discussion – I discuss the scheme in this context. STAMPede is an important scheme, because it summarises and unifies many prior approaches.

The final version [76] uses an extended coherence protocol for managing speculative memory versions in the cache system. It utilises compiler support to minimise hardware complexity, and builds on the respective strengths of the compiler and hardware: the compiler has a static view of the whole program, while hardware has insight into local dynamic behaviour, as well as run-time data. The authors claim allowing arbitrary memory accesses as a novelty, even though several previous schemes [43, 50, 68, 74] had already supported this [17, 93]. Explicit synchronisation is used to handle dependences found by the compiler, and stacklets [24] (also known as *cactus stacks*) eliminate stack dependences. With stacklets, instead of all tasks sharing a single stack, they each have their own private, heap-allocated copies of the top of the stack, where they can push and pop new call frames without creating a cross-task dependence. In-order thread(let) commit is implemented using the *homefree token*, marking the oldest thread. Only the thread holding this token can commit, and the token is passed to the next thread when the current owner successfully commits. Although the authors of Swarm later criticise this serialisation as a bottleneck [37], this is likely due to the significantly higher number of threads in Swarm (1024 instead of 2-16).

STAMPede achieves 16% speedup on general-purpose applications using a private-cache 4-core CMP design. The limiting factors are coverage (44% of run time), overheads in inter-core communication (including increased L1 miss rates), high thread spawning cost and communication delays and high squash rates due to a combination of slow communication, large threads and coarse conflict checking (performed at cache-line granularity). Out of the time spent inside parallel regions, more than a third (36%) is spent on failed speculation (11%), D-cache misses (17%), synchronisation and spawning (9%) in the private-cache implementation. If the 4 cores share a cache, the time spent on D-cache misses decreases by 5%, but the benefits are negated by a jump in time spent on failed speculation (27.5%). Additionally, load balancing also presents an issue for some benchmarks.

Out of all CMP-based TLS schemes that do not rely on significant deviations from the programming model (and give a reasonably detailed implementation on a large out-of-order baseline processor), this result is the state-of-the-art at the time of writing of this thesis, to the best of my knowledge.

2.3.2 Swarm project

To cut down on the overheads of TLS (as detailed above), Swarm [37] takes a more radical approach than most TLS schemes. The authors identify that – in trying to amortise away the overheads of task spawning, queuing, commit and scheduling – CMP-based TLS schemes

opt for large tasks, consisting of thousands of instructions. This in turn leads to high amounts of speculative state, as well as frequent runtime conflicts. The proposed solution is to cut the overheads both by moving task management (including queuing and scheduling) into hardware, and to use small tasks. To expose sufficient parallelism, the task window is expanded from 2-16 to up to 1024 tasks. This in turn requires high commit bandwidth, which is achieved using a distributed commit mechanism and efficient conflict detection using Bloom filters. Furthermore, to take advantage of the granular division of code into tasks, more sophisticated selective aborts are also incorporated: the only tasks aborted are ones that read stale data (produced by an aborted epoch) and their successors.

To reduce aborts further, the concept of successors is redefined. Instead of a linear chain of tasks, Swarm defines a tree (or forest at runtime) structure. This model is easily applicable to algorithms with ordered irregular parallelism, which is a common pattern. On the other hand, significant modifications are required to the programming model in order to exploit the benefits of this approach.

Swarm uses a distributed model for scheduling and conflict detection, with eager version management. Speculative state is exposed to the (coherent) cache system, and aborts are performed using an undo log.

Fractal

Fractal [78] introduces nested parallelism into Swarm using ordered and unordered *subdomains*. Each subdomain can contain tasks and further subdomains as its children. From the parent's view, these execute as atomic transactions. Ordered domains use (compiler- or programmer-assigned) timestamps to define a total order of tasks explicitly. For unordered domains, timestamps are assigned at runtime (when each task is dispatched), by hardware. Fractal defines a mechanism to *zoom in* at runtime when appropriate, thus simultaneously utilising the innermost N layers (they use $N = 4$) of parallelism (e.g. the N innermost profitably parallelisable levels of a loop nest).

As before, this abstraction is very expressive and powerful, but it changes the programming model significantly.

T4

T4 [92] goes even further than Swarm in terms of task sizes, introducing 'tiny tasks of tens of instructions'. The main goal is to reduce squashing as much as possible. This is achieved by eliminating unnecessary parent-child relationships, and splitting tasks further.

Most importantly, three aggressive optimisations are implemented. Firstly, contentious accesses (instructions which are likely to cause conflicts) are moved to their own tasks. In practice, most memory accesses will be in a separate task from arithmetic instructions computing the address and data for the access. Secondly, task spawning is separated from worker tasks. Thirdly, the shared stack is eliminated in favour of continuation passing style, and heap-allocated 'stack'-frames. Some further tricks, such as locality hints (mapping memory accesses to the same location to the same tile using address hashing) also improve performance.

Additionally, T4 gives a full toolchain – including a fully-fledged compiler – for Swarm, and it evaluates over general-purpose benchmarks. Out of the 20 benchmarks in the C/C++ portion of the SPEC INT 2006 benchmark suite, T4 fails to compile 10 due to the significant programming model changes (specifically the changed structure of the stack, difficulties with exception handling). Of the remaining 10 benchmarks, 5 favourable ones achieve 19× geometric mean speedup using 64 cores compared to the program running on only one of the cores, while the remaining 5 struggle due to high conflict rates (manifesting as failed speculation and throttling), leading to only 1.4× speedup using 4 cores, and lower for higher core counts (1.1× for 16 cores, not reported for 64 cores). Averaging over all 20 benchmarks (using optimal number of cores), the geometric mean speedup is only 2.3× (5.2× for benchmarks that compiled successfully) over the single-core baseline.

Discussion

Swarm is an aggressive, novel technique, which achieves impressive scalability and good speedups. It features a detailed, well-optimised implementation, with a comprehensive list of extensions. However, comparing a proposed commodity TLS implementation against it is not fair.

Swarm requires significant changes to the programming model and the architecture. This is well exemplified by the fact that half of the benchmarks in the SPEC CPU 2006 suite could not be compiled. While the authors claim it would be possible to compile these workloads, they admit it would take significant engineering effort. Since many of these programs are from the non-numeric, historically harder to parallelise integer suite, and the extensive changes required to enable the complex stack usage and exceptions these benchmarks would presumably introduce additional overheads, it is reasonable to expect low speedups (or even slowdowns) on these benchmarks. The effect of these additional transformations on the rest of the benchmarks is also unknown. In summary, quoting the geometric mean speedup as 2.3× seems fair, until an improved compiler is released. Since this is over a custom 64-core system, system utilisation is extremely low.

Additionally, the architectural impact of Swarm would be very significant. In particular, speculatively stored values are exposed to the coherent memory system (due to the use of an undo log for implementing aborts). In current systems the memory consistency model does not allow this to happen, so changes would be required to this. Additionally, special synchronisation instructions would need to be added around any use of special memory regions (e.g. memory-mapped files, device memory, I/O controller addresses). Furthermore – while the authors hint that some programs could use only a partition of the system (e.g. 16 or 4 cores) – in reality multithreading would also be very challenging for the same reason. Specifically, Swarm's model is not compatible with the C++11 memory consistency model out of the box, and implementing compatibility is not straightforward. Without significant further innovation, existing multi-threaded software would likely need significant programmer modifications for stable results following a (new) well-defined consistency model, which does not publicly exist at the time of writing.

Considering the above, we should perhaps consider Swarm as a highly-threaded accelerator architecture, rather than a general-purpose CPU proposal. In this case, the area overhead

compared to a single CPU is 64×, leading to 2.3× geomean speedup (5.2× over compiling benchmarks). Perhaps a similar CPU could be designed from the ground up, but it would require an extreme amount of engineering effort. In this thesis, I argue that a less disruptive method, even if maximum speedups are significantly reduced, is more likely to be adopted by industry in the foreseeable future.

2.4 TLS in a simultaneously multi-threaded pipeline

Shortly after the initial (processing unit based) TLS papers were published, in 1998, Tullsen et al. [84] proposed simultaneous multithreading (SMT) as a way to increase utilisation in the increasingly wide superscalar pipelines of the future.

SMT (or Hyper-Threading in Intel's terminology) maps two (or more) architectural threads to each processor core. Architecturally, these are separate execution contexts, each with its own architectural state (i.e. architectural registers and program counter). They are exposed through the operating system as virtual cores, thus the technique is fully hidden from user programs, which simply see twice as many available cores. However, the micro-architecture maps two (or more) contexts into a single pipeline. While some structures – such as the register rename map, program counter and fetch address – need to be duplicated to track and update architectural state of each thread correctly, others can be partitioned or dynamically shared.

Most importantly, functional units (e.g. integer ALUs, load pipes), the issue queue, pipeline bandwidth (e.g. decode/issue/commit width), load/store bandwidth, physical register file and free lists can be shared dynamically. Each instruction that flows through the pipeline is tagged with a context ID, to facilitate separation between contexts, and to ensure any instruction-related events (e.g. instruction commit, exception or branch misprediction) update the corresponding thread's state.

As noticed by the papers mentioned in this section, these special features of SMT have some advantages when running closely-related (speculative) threads efficiently. Firstly, the close arrangement and the shared L1 cache make inter-thread communication very efficient, and it reduces coherence traffic on false sharing of cache lines too. Secondly, limiting speculative state to a single core allows it to be hidden from the rest of the system, thus giving significantly more freedom to implement a consistency model, as defined by the architecture.

Let us now look at the individual proposals in this space. I focus on the novel idea(s) introduced for handling speculative state and the individual limitations preventing the direct application of these schemes to modern high-performance chips.

2.4.1 Threaded Multipath Execution

Wallace et al. [88] notice these advantages, and they propose a scheme that – upon encountering a hard-to-predict branch – launches both targets in separate threads inside a SMT pipeline (so long as sufficient contexts are available). Once the branch outcome is resolved, one of the threads is discarded and recycled. The already low speedups measured on a processor with a far shorter speculative window (and thus fewer active branches) than today's CPUs mean

this scheme is only interesting from a historical perspective. Indeed, later measurements [61] predict a slowdown over the newer SPEC CPU 2000 suite.

2.4.2 Dynamic Multithreading Processor

Akkary and Driscoll [1] (DMT) proposed to use SMT threads to perform low-overhead thread-level speculation. Their proposal adds a second-level instruction window to allow speculative instructions to complete and leave the main (first-level) re-order buffer. Recovery from conflicts is performed using granular (per-instruction) re-execution. To do this, the offending instruction and its dependences are reloaded from the second-level instruction window, and re-executed.

Conflict detection and data versioning are implemented with the help of the load-store queue (LSQ). Speculative memory load instructions are allowed to execute (reading memory or previous in-flight stores) and complete, but they are kept in the load queue until their final (architectural) commit, so that their address can be checked against possibly conflicting store addresses. This mechanism is similar to load-to-store reordering and squashing in modern pipelines, except for the addition of the second-level instruction window. Speculative stores, on the other hand are kept in the store queue (or store buffer), thus their value can be forwarded to future loads, but they only affect cache state once they fully commit.

This scheme has the advantage of not exposing speculative state to the memory system (thus maintaining compatibility with existing memory models, and multi-threaded programs), however the LSQ limits speculative state. Furthermore, the sophisticated squash and recovery mechanism proves to be too expensive for large threads. The original paper measures 15% improvement over SPEC CPU 95, rising to 30% if an extra I-cache port is added, however these numbers are challenged by the authors of the IMT (described in the next subsection), who measure the impact of the same scheme as a 12% slowdown over SPEC CPU 2000, even with very generous assumptions.

2.4.3 Implicitly-Multithreaded Processors

Park et al. [61] proposed the implicitly multithreaded processor (IMT) to cut the hardware requirement compared to DMT by removing the granular abort mechanism and the need for the second-level instruction window. They still use the LSQ to buffer speculative operations, but they implement aborts using a checkpointing mechanism. They save the register rename map on thread launch (copying appropriate register mappings from a *master rename table* to a *local rename table*), and simply roll back to this if any conflicts are noticed. They use the Multiscalar compiler to set up implicit threading, and implement three key optimisations to gain performance. Firstly, the fetch policy considers predicted resource usage and thread ordering, favouring fetch from threads that are less likely to abort (due to resource constraints or deep speculation). Secondly, they implement *context multiplexing*, where multiple threads are mapped onto the same context. This is implemented by leaving gaps in pipeline structures (e.g. ROB, LSQ) based on the estimated resource requirements of predecessor threads mapped to the same context. Finally, they perform the checkpointing in the background, by only using spare bandwidth for copying rename maps. Without these optimisations, performance gains are only nominal, whereas with them, the authors claim

24% geometric mean improvements over SPEC CPU 2000. The authors also reproduce the previous two techniques (sections 2.4.1 and 2.4.2), finding performance regressions over SPEC CPU 2000 with a more realistic superscalar baseline.

However, there's evidence these improvements would not hold up on modern systems. For integer benchmarks, the baseline (a large 8-wide OoO superscalar with an active window of 1024 instructions) claims an IPC of only 1.4, which is significantly weaker than even smaller modern systems can achieve[2]. This is despite assuming a low-latency main memory (80-cycle latency). Furthermore, over a third (34%) of the elapsed clock cycles in the baseline are taken up by data dependence stalls. This implies that there are insufficient independent instructions in the pipeline. Even so, this is significantly reduced (to 24%) by the naïve version of the scheme presented in the paper, which only maps a single thread containing $10-20$ instructions onto each of the 8 contexts, leading to a maximum look-ahead of $80-160$ instructions. This should be lower than the effective look-ahead of the baseline machine, which has an active window of 1024 instructions. Even the optimised version, with a multiplexing factor of $3-6\times$ would only support a total look-ahead of $240-960$ instructions. These factors suggest that the main improvements likely come from a weak front-end, which is unable to fill the out-of-order window effectively, even on benchmarks without particularly complex branching. Modern machines, with highly-accurate branch predictors and decoupled front-end pipelines [36] do not typically exhibit such frontend-bound behaviours on SPEC CPU benchmarks [91].

2.4.4 Packirisamy et al.

Packirisamy et al. [59] notice that previous schemes – relying on LSQs for buffering speculative state – struggle to support larger threads, leading to limited gains. They propose a multi-versioned L1 cache to hold speculative state, and describe in detail how to modify the cache coherence protocol to implement this. They describe a simpler 2-thread version, relying on 9 extra bits per cache-line (one *speculatively loaded* (SL) bit, and 8 *speculatively modified* (SMi) bits, one per data word), as well as a more complex 4-thread version, relying on 16 extra bits (four SL and four *owner* bits, one per thread, as well as 8 SMi bits). Even though they claim to have a working compiler and an automated pipeline, they only evaluate over 5 benchmarks from SPEC CPU 2000, without explaining how these were selected. Over these 5 benchmarks, the 2-thread scheme achieves 10% improvement, while the 4-thread scheme scores 15%, the latter of which provides a slight improvement of 3% over an LSQ-based TLS implementation. The improvements over the LSQ-based approach come from buffer overflow stalls, as expected. The main improvements compared to the baseline come from decreased cache misses. This is somewhat surprising given that the cache protocol described leads to an L1 cache miss every time the oldest thread accesses a cache line that has been speculatively modified, but presumably this is due to the prefetching effect of the speculative threads. However, the simple trace-based evaluation and the use of limited workloads likely puts the performance results within the margin of error, and so this paper is most interesting for its cache coherence protocol proposal.

[2]We measured a geometric mean IPC of 1.54 over SPEC CPU2006 (which has more complex data and control flow) using our Intel® Xeon® W-2195 system (4-wide OoO superscalar, released in 2017).

2.5 Versioned caches

One of the crucial aspects of performing thread-level speculation is to be able to hold different (speculative) versions of the same memory locations [17, 83]. Similarly, holding and handling different versions of memory, and publishing multiple updates simultaneously, is also a problem hardware transactional memory extensions have to solve [34, 38]. Here, I discuss the state of the art in supporting versioned data in the cache system.

Multiscalar [74] was the first TLS scheme to propose a solution by providing the address resolution buffer (or ARB). This structure resides on the memory side of the interconnect, and it simply buffers all memory operations (reads and writes). It would perform conflict checking, and upon committing a speculative thread it would write back the speculatively-written values to the memory system. As a system with a single processor core, designed for thread-level speculation but not multithreading, the ARB can track and resolve all dependences and squash threads appropriately. Its design and operation is similar to today's load and store queues, which can handle speculative loads and stores, possibly observed out-of-order, and find and resolve conflicts by squashing offending loads, before eventually writing back updates to the main memory system in-order. Note that such non-atomic writeback would not be safe on a multi-core design which supports traditional multithreading, as a (non-speculative) write from another core could conflict with a thread partway through its commit, leaving it in an unabortable but uncommittable state. Additionally, the scalability of the ARB is poor, as it collects all memory accesses in the system at a single point, on the memory side of the interconnect.

Later work proposes a speculatively versioned cache (SVC) [25], which brings speculative data into the private caches of processing units (or cores). This work expands the cache coherence protocol to support this. Cache lines may be tagged as speculatively-loaded and/or speculatively-modified. Furthermore, each cache line links to the next version (in another cache) through the version control list (VCL). The SVC work proposes multiple optimisations to improve performance, including efficient commit (EC), which allows speculatively-written cache lines to be tagged as *committed*, meaning that they will respond to coherence requests, and can be written back lazily. This scheme provides an efficient method for cache-line-level memory versioning and disambiguation, and it is very similar to the schemes used by later TLS schemes, such as STAMPede [76]. Its two main limitations are the coarse granularity (cache-line level), which leads to false sharing and the inability to handle write-after-write conflicts without squashing, and the fact that – just like the ARB – it does not support traditional multithreading on top of TLS due to coherence issues during commit.

Hardware transactional memory (HTM) [34] introduces atomic *transactions* [26]) natively into the memory system. To support atomicity, hardware transactional extensions need to implement multi-line atomic operations, including reads and writes. Unlike TLS-only systems, HTM needs to integrate with regular multithreading, and observe the rules of coherence and consistency in the system. The common approach [34, 38] for implementing this is to build on top of the cache coherence protocol. Lines read by a transaction are brought into the private cache in the shared state, and those written are brought in in a private (modified or exclusive) state. Any snooped bus messages transitioning away from these states either cause the transaction to fail, or cause the requester to fail (if it is also a transaction).

Commit is performed locally in the private cache, appearing to be instantaneous from the outside. This may be implemented [34] using a small fully-associative *transactional cache* to hold transactionally-written lines, and marking them as *non-transactional* in a single cycle upon commit. Then, these respond to snoop requests from other caches, and they are gradually evicted due to the transactional cache being full, or the lines being requested by other caches. The cache designs used for hardware transactional memory are efficient at providing cacheline-level multi-line atomic accesses, and these designs have even seen adoption by industry (see Intel's TSX and IBM's TM). As with TLS versioned caches, conflicts are detected on a cacheline granularity. Additionally, due to the cost of the fully-associative cache, there is an upper limit on the size of transactions.

2.6 TLS summary

While a high number of TLS schemes have been developed, evidence is insufficient to justify wide-scale adoption. Although some papers achieve impressive gains, the numbers are sometimes inconsistent, and they often rely on significant assumptions, therefore gains may or may not translate well into practice. Furthermore, compatibility with today's microarchitectures and architectures remain open questions. Substantial changes to the architecture (e.g. memory model), or the programming model would often be required, and this adds significant hurdles – as well as additional risks – for adoption in the field. Furthermore, most schemes use high-level trace-based evaluation techniques (which have a large margin of error), and the targets are much smaller cores than today's state of the art. Therefore, a re-evaluation and new combination of ideas is necessary to realise and update these gains.

In this thesis, I propose schemes that can be added to a modern system, with minimal, optional (hint-based) additions to the architecture, as well as localised, feasible adaptations to the microarchitecture, and no changes to the programming model. These properties make the proposals more likely to be widely adopted, and thus to successfully make a difference in the field. They may also serve as a gateway to adding more ambitious schemes in the future, with a higher speedup potential.

2.7 Pipeline parallelism

Another way to expose parallelism at a similar granularity is to break a program into pipeline stages that can communicate with each other.

First, decoupled access-execute (DAE) [73] architectures proposed to fully decouple memory accesses from arithmetics and control flow calculations (i.e. execution) to take advantage of their different characteristics. As advances in out-of-order execution and superscalar machines became better at hiding access latencies, the gains from this shrunk.

New approaches, such as decoupled software pipelining (DSWP) [66] increase the size of tasks (or pipeline stages). DSWP partitions loops into multiple stages. As a common case, it decouples the loop variable update or traversal (for example, extracting elements from a data structure) from the work within the loop body (such as performing an operation on each element). Each core in the processor runs all invocations of a single stage in order. Stages

within a single iterations communicate via new *produce* and *consume* instructions, through the architecturally visible synchronisation array [66], which is implemented in hardware for efficiency. Automatic thread extraction [58], and speculative pipeline parallelisation [85] are also shown to be possible, although the latter does not seem to have a significant effect on overall performance. DSWP introduces significant overheads due to the use of a multi-core setup and custom synchronisation hardware. Despite this, only relatively modest gains are demonstrated.

A wealth of other work improve on DSWP [13, 14, 28, 79, 80], by adding load balancing, support for more complex pipeline stage structure and efficiency improvements. PIPETTE [55] brings the improvements together and adds additional features to expand the scope and increase the efficiency of pipeline parallelism. Crucially, it brings pipeline parallelism into a single core, thus reusing core resources and achieving the gains without the extensive overheads of a multi-core machine. The focus is on increasing core utilisation, similar to the proposal in this thesis. Whilst good gains are shown on some workloads, this is achieved by rewriting programs to use PIPETTE's pipeline abstraction. The authors acknowledge that transforming programs into the new abstraction will likely never be automatic, and instead argue that manual transformation of library code would be feasible and may bring significant overall improvements. This restricts the applicability of the technique. It is worth noting that PIPETTE is not necessarily mutually exclusive with the proposals in this thesis. A processor with capabilities for both is feasible, although running both schemes at the same time is not investigated in this thesis, and it requires further work.

2.8 Direct inspiration

Now we have explored past TLS and pipeline parallelism work, I present two papers that inspired and heavily influenced my research. Tapir [70] inspired the format of the basic hint instructions and it served as a basis for the compiler. Loopapalooza [93] was (co-)developed by colleagues at Arm, and it served as the starting point, as a limit study.

2.8.1 Tapir

Tapir [70] – a compiler paper – embeds parallelisation instructions into LLVM's internal representation (IR), thus enabling more sequential optimisations. This is novel compared to standard approach of implementing parallelisation suites fully in the compiler's front-end, which ends up generating opaque function calls in the IR, thus blocking almost all sequential optimisations. By embedding annotations directly into mid-level IR, rather than hiding them, the compiler mid-end can understand them better. This enables most sequential optimisation passes can work either out-of-the-box or with minor modifications.

To make the embedding feasible, Tapir uses the asymmetric fork-join paradigm, which is very expressive, and it has some nice properties. *Fork-join parallelism* divides a program into sequential and parallel segments, exploiting parallelism between different threads in each parallel region. *Asymmetric fork-join* defines a strict ordering between threads, thus restoring the concept of a global program order between any pair of instructions or blocks (in any thread). Having a well-defined program order is crucial, as it leads to much closer

alignment with sequential IR. In particular, conditional jumps forward to a future point lead to very similar semantics compared to asymmetric fork-join parallelisation.

Tapir introduces three new instructions: *detach*, *reattach* and *sync*. The detach instruction (in a block *A*) takes two successor blocks, *B* and *C*, as arguments. Execution from block *B* terminates upon reaching a reattach instruction containing block address *C*. The code region between the start of *B* and the reattach instruction is called the *detached* block, and the region starting at *C* is called the *continuation*. The detached block and the continuation can execute in parallel, but their roles are asymmetric. The detached block terminates with a reattach, whereas the continuation leads into the rest of the program. In program order (and if not parallelising), the detached block comes before the continuation, followed by the rest of the program.

The continuation may spawn further workers by invoking more detaches, until it eventually reaches a *sync* instruction. This marks the boundary of the parallel region, acting as a barrier: any instructions past it are able to observe the effects of any instructions prior to it.

To model this in IR, take the sequential program that executes the detached region *before* the continuation. That is, it branches from the detach instruction (at the end of block *A*) to block *B*, and then from the reattach instruction to block *C*, continuing to through the rest of the program. Then, add a (conditional) control-flow edge from the detach to the continuation, referring to control flow in the second thread. All of these edges are also specifically marked to prevent optimisations from moving instructions in a way that would create new determinacy races. A race occurs if both the detached region and the continuation access a location, and one of those accesses is a write. Tapir's open-source implementation shows the required changes to optimisation passes.

The compiler (developed by my colleagues) for the architecture described in this thesis relies on Tapir's IR instructions (with minor modifications), but it adds a new back-end to Tapir. The three instructions are lowered all the way into the instruction set. Section 3.2 describes the exact semantics, as well as the additional edge cases that arise at this new, lower level of abstraction.

2.8.2 Loopapalooza

Loopapalooza [93] characterises loops based on their potential for being parallelised. The authors define three different theoretical parallelisation cost models called DOALL, PDOALLand DOACROSS, and derive limits for the amount of speedup achievable (with unlimited resources) using each model. I rely on this characterisation throughout this thesis, and use the limit study results to justify the existence of significant potential parallelism.

Loopapalooza assumes that false (WaW, WaR) and structural dependences are possible to handle, hence the authors concentrate on true (RaW) register and memory conflicts. I give a classification (based on Loopapalooza) of dependences in table 2.1, as this is interesting, and provides valuable definitions for discussions later in the thesis (particularly Chapters 4 and 5).

Computable register dependences are assumed to be eliminated by the compiler (e.g. by computing them from the iteration number, locally in each iteration). The rest of the dependences are examined under three different execution models:

Register	**Computable** (via scalar evolution, e.g. for induction variables)	
	Non-computable	**Reduction accumulators**
		Predictable
		Unpredictable
Memory	**Frequent**	**Static** (known at compile time)
		Dynamic aliasing (frequent)
	Infrequent	**Static** (infrequent control path)
		Dynamic aliasing (infrequent)

Table 2.1: Cross-iteration dependence classification based on Loopapalooza. Computable dependences are determined using LLVM's (mutual) induction variable analysis, predictibility is determined using a fixed combination of several value predictors, and static dependences are determined using LLVM's alias analysis. A frequent dependence is one that occurs in 80% of iterations or more.

DOALL Parallelise all iterations, but only for loops with no memory dependences and only computable register dependences.

PDOALL Partial-DOALL. Only computable register dependences, but there may be (true) memory dependences between some iterations. Execute iterations without true dependences in parallel, like DOALL. However, if an iteration i consumes a value from a predecessor j, delay i and all its successors until j finishes. This simulates (failed) speculation.

DOACROSS Simulate static synchronisation instructions inserted by the compiler optimally: find the smallest fixed δ such that starting each iteration δ cycles after the last one eliminates all true dependence violations (including non-computable register and all memory dependences).

The investigation is performed using different relaxations for non-computable register variables (e.g. lowering or prediction), and different assumptions on what function calls are allowed. The finite nature of pipeline resources is not modelled, and the analysis relies on LLVM passes, and LLVM instruction counts. Predicted upper bounds on speedup are promising: 'realistic' setups (called `reduc1-dep2-fn2` in the paper) predict $1.2\times$ speedup for SPEC INT2000, $2.0\times$ for for SPEC INT2006, and much higher ($6-20\times$) for numeric benchmarks (EEMBC, SPEC FP2000 and FP2006) for the PDOALL model. The DOACROSS model performs significantly better due to high coverage, at $4.6\times$ and $7.2\times$ speedup for SPEC INT2000 and INT2006 respectively and it delivers $21-50\times$ speedup for numerical benchmarks.

These numbers are very encouraging, although the setup only tests the limits of parallelism: this will be much further restricted once limited resources, per-iteration costs and further compiler limitations are factored in. However, there may also be some opportunities for improvement. Firstly, better predictors may achieve anywhere up to $2.0\times$ and $2.6\times$ speedup (`reduc0-dep3-fn3` setup) on the integer benchmarks with PDOALL alone. More sophisticated compiler analysis – able to identify more computable variables – may also be used for similar gains, although this is questionable, due to non-computability and the lack of runtime data. Secondly, if we allow iterations to observe values written by earlier itera-

tions before those earlier iterations complete, then the PDOALL model may behave largely similarly to the DOACROSS model for memory conflicts (and lowered register conflicts).

These optimisations and limitations are discussed later in this thesis.

Chapter 3

Base implementation

This chapter describes an in-core, hint-based thread-level speculation scheme, based on the principles discussed in the previous chapters. The scope here is to make a fully functional scheme, which is compatible with modern high-performance (out-of-order, wide superscalar) processor pipelines, solving all arising practical issues concerning correctness and adoptability. Further to this, I also analyse the performance, and identify the bottlenecks in this design. While I address trivial bottlenecks as they arise, we will see that some more complex bottlenecks also exist. These will be addressed in the next chapter, as they require a significant redesign, and novel techniques to overcome.

I present an architecture and microarchitecture here, as well as a summary of compiler considerations. My main contributions are the microarchitecture and the corresponding analysis. The architecture was co-developed with researchers from Arm. The prototype compiler was implemented by colleagues at the Computer Laboratory, with input and contributions from Arm.

3.1 Overview

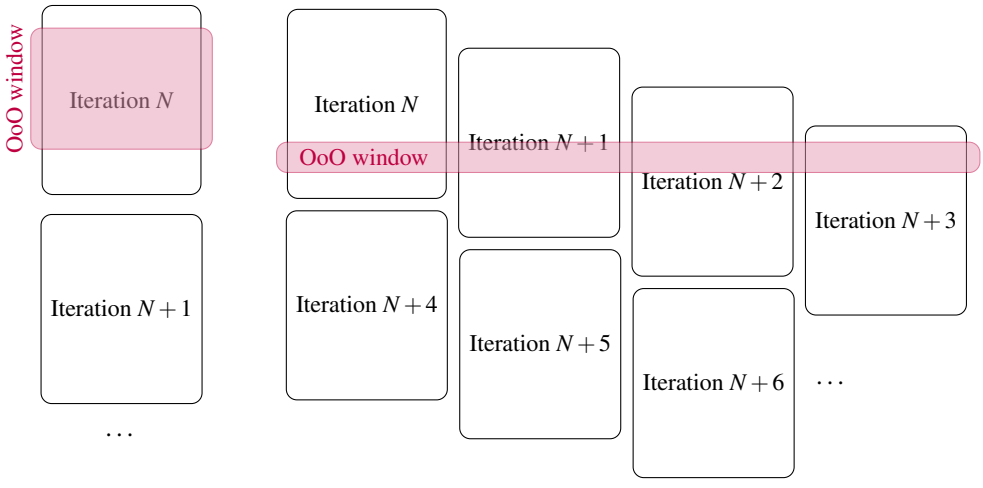
The compiler exposes potential parallelism using asymmetric, ordered fork-join hints. The hints work similarly to Tapir's [70] IR instructions. They mark the loop bounds and induction variable updates. The microarchitecture can either *take* the hints to exploit parallelism, or *ignore* them to execute the program sequentially. Such decisions can be made either statically (e.g. a low-performance or older system can always ignore these hints), or dynamically (e.g. ignoring hints for unprofitable regions). These hints guarantee certain *control-flow* and *register dataflow* properties (see Section 3.2), but crucially do *not* guarantee independence of memory operations. A sophisticated compiler can try to eliminate or reduce memory dependences on a best-effort basis, but this is not required for semantic correctness. We chose this permissive approach in order to reduce the mandatory burden on the compiler, allowing it to side-step the complexity and computability issues of precise alias analysis in mainstream languages like C and C++.

Parallelisation using hints If hints are taken, the microarchitecture internally creates multiple threadlets (see Section 3.6), each executing a different quasi-independent program

epoch[1]. The oldest threadlet executes as usual, *architecturally*, while other – *speculative* – threadlets buffer their memory updates in the *speculative state buffer* (SSB, see Section 3.10.2), and record necessary information (e.g. memory read and write sets) to check for conflicts with predecessors (see Section 3.11). If a conflict is detected, the younger threadlet is *squashed* and restarted. When this happens, any successors of the squashed threadlet are also squashed and discarded. Multiple threadlets can run simultaneously, in the same processor pipeline, using lightweight simultaneous multithreading (SMT) support (see Section 3.4). Pipeline resources are shared between threadlets, with priority given to older, less speculative threadlets (see Section 3.7). Speculative threadlets are committed to architectural state in threadlet order, once the lack of conflicts with predecessors is demonstrated by the conflict detector, exposing their effects (i.e. memory reads and writes) to the coherent memory system, while respecting the architecture's memory consistency model (as detailed in Section 3.10.6).

Source of performance gains As shown in Figure 3.1, mapping these epochs to multiple (speculative) microarchitectural *threadlets* splits the out-of-order window into multiple (smaller) independent sub-windows, which helps to expose parallelism in three ways. Firstly, the sub-windows each cover a disjoint segment of the dynamic instruction stream and these segments are typically not contiguous. Due to the gaps between the segments, they collectively cover (parts of) a wider region of the instruction stream than a single, contiguous window with the same number of total instructions would. Crucially, the sub-windows will – together – contain instructions from multiple (quasi-independent) iterations, thus enabling more reordering, due to infrequent memory conflicts and no register conflicts. Secondly, the impact of branch mispredicts are limited to a single threadlet (or sub-window). The other threadlets can continue running while the mispredict is resolved. Thirdly, this also allows for

[1]Epoch: a contiguous slice of the dynamic instruction trace. The instruction sequence executed by a threadlet.



(a) Sequential

(b) Parallel (one column per context, no conflicts)

Figure 3.1: A single contiguous out-of-order window (sequential execution) may not expose cross-iteration parallelism. Splitting it into sub-windows (parallel execution) can help.

out-of-order fetch, rename and dispatch, because register and control dependences are cut during execution time, which can increase the throughput of the front end pipeline (which is normally forced to run in-order).

Clearly, this system is a co-design of architecture, compiler and microarchitecture. The next sections dive into the operation of each component, followed by a performance analysis of the scheme, and a discussion of the results.

3.2 Architecture

This section describes the architecture, making repeated reference to its intended (microarchitectural and compiler) usage. Doing so, we can explore all necessary architectural modifications, as well as optional modifications that significantly ease code generation. As argued in previous chapters, the minimal and optional nature of the modifications here are crucial. Apart from losing backwards compatibility, major and disruptive changes would inevitably raise the deployment barrier, due to extra implementation and verification effort, and the need to ensure inter-operability with other complex microarchitectural techniques and architectural extensions.

3.2.1 Overview

The core architecture relies on simple hint instructions in order to aid the microarchitecture to implicitly parallelise regions of code, such as suitable loops[2]. The hints do not change program semantics. Even without hints, the microarchitecture could – theoretically – already internally perform all the reordering described in this chapter, while preserving the illusion of sequential execution. However, the compiler is better suited to perform certain program analyses and transformations, due to its more global and thorough view of the program and more generous memory and computation budgets than available at run time. The hints serve as a way to facilitate the compiler-microarchitecture codesign by providing an architectural way to communicate this information. This section discusses the parallel semantics, that is the additional information encoded by the hints. Furthermore, I outline on a high level how a microarchitecture could utilise the knowledge.

The hints encode the bounds of *parallel regions* and program *epochs* within them. A *parallel region* is the part of the program targeted by a collection of parallelisation hints. Each hint corresponds to a single region, defining the strategy for parallelisation. Regions are either disjoint, or one region may be fully nested within another, in which case the inner region needs to be contained between two subsequent hints of the outer region. At run time, each region is divided into multiple epochs. An *epoch* is a collection of dynamic instructions that may be parallelised with other epochs. The hints denote suitable points for forking future epochs such that cross-epoch data flow is likely limited, and thus parallelisation is likely to succeed.

It is important to note that the hints cannot change sequential semantics even in the presence of cross-epoch dependences, therefore the microarchitecture needs to either obey

[2]The terminology here only discusses loops for simplicity, but parallelising other types of regions is possible, as discussed in Section 3.2.6.

dependences, or check for them, and discard any work that was performed based on stale data. Thus, there are no hard limitations on cross-iteration data dependences that the compiler (or programmer) is allowed to emit.

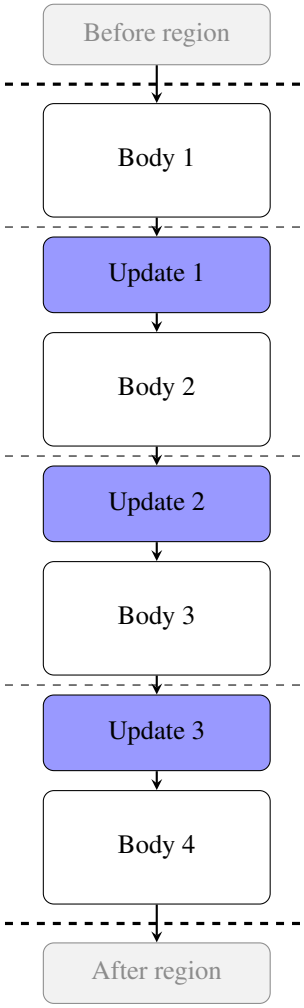


Figure 3.2: A simple parallel region executed sequentially. Epochs (separated by dashed lines) start with an *update* section (containing known dependences) and a parallel *body*.

However, note that some cross-iteration (and thus cross-epoch) data flow is likely to always exist. For example, even for simple loops, the induction variable update will form a cross-iteration dependence. As we will see in Section 3.3, compilers are good at analysing register data flow, and this information is useful to the microarchitecture (to avoid dependence violations). Therefore, the hints provide a way to identify code that produces the input values to the next epoch. Each epoch starts with an *update* section, before launching the next epoch and executing the parallel *body* before terminating.

The same logic could be used for cross-epoch memory dependences too, but this is more challenging for both the compiler and the microarchitecture. Compiler alias analysis is known to be difficult, costly and undecidable due to the lack of run-time information, leading to overly conservative outputs [51]. The microarchitectural aspects of forwarding memory values between parallel epochs is discussed later, in Section 4.2. Thus, the prototype compiler puts all sources of register data flow into the update section, but it leaves memory dependences in the body.

Figure 3.2 shows the dynamic trace of a sequential loop chopped into epochs (dashed lines), with each epoch[3] separated into two sections, as shown. Figure 3.3 shows how this enables parallelisation. After entering a region, each epoch starts with an induction variable update section, which updates all variables that cause inter-iteration register dependences (such as induction variables). Following this, the next epoch can launch and execute in parallel with the body of the current epoch (and any previous epochs). The successor inherits the register state of the current epoch, which correctly handles any cross-epoch register dependences where the producer is in the update section. The end of the region is annotated to limit the scope that compiler analysis needs to consider (this is discussed more in Sections 3.2.4 and 3.3), and to facilitate low-cost control speculation when launching epochs; if an epoch reaches the end of the region, then any successors it has launched can be discarded.

[3]The first epoch does not have an update section, as we define the region to start at the start of the first body, as all values produced before the region are automatically available inside the region.

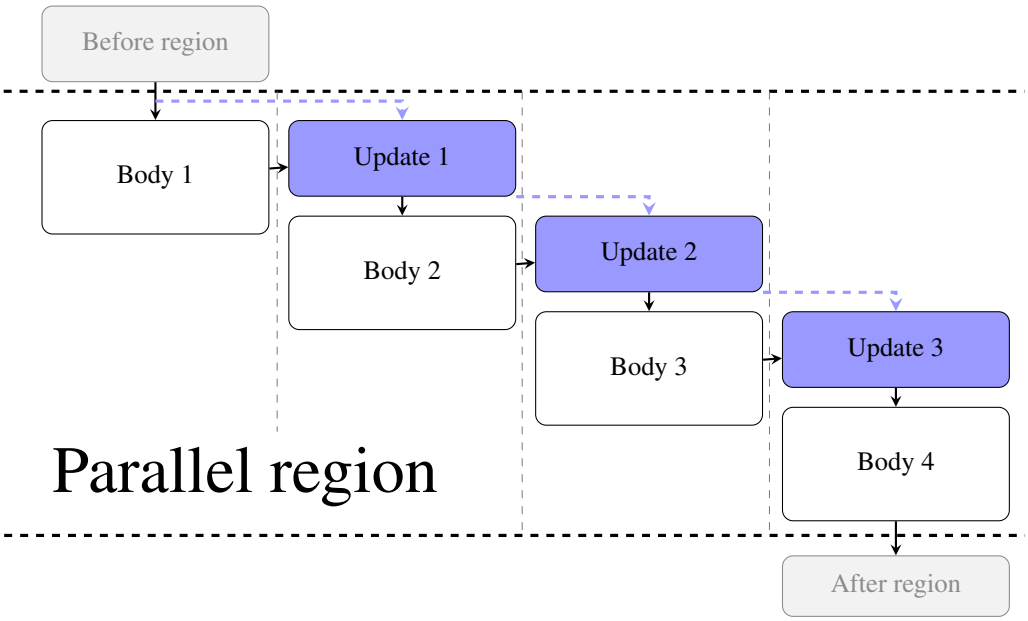


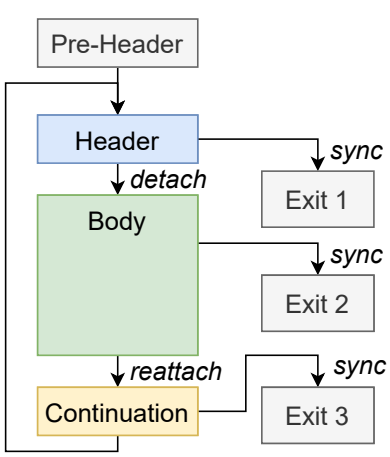
Figure 3.3: Simple parallel region executed in parallel, one column per epoch. The black arrows show program order, along which sequential semantics are maintained. The blue dashed arrows show the fork points. Data flow along the horizontal black edges creates a hazard.

3.2.2 Parallelisation hint instructions

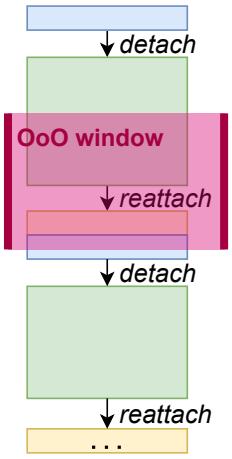
The core architecture introduces three new hint instructions, inspired by Tapir (Section 2.8.1). These are `detach`, `reattach` and `sync`. The `detach` marks the start of a parallel body, `reattach` marks the end of the body, and `sync` marks the end of the region. Equivalently, `detach` marks the fork point where we detach the next epoch, `reattach` marks the end of the current epoch (and the start of the next one), and `sync` marks the synchronisation barrier at the end of the region.

Figure 3.4(a) shows how the above structure may be applied to the CFG, that is, the static view of the program. Since the initial update section from Figure 3.3 is now split into two parts by the back edge, let us denote these two parts as the *continuation* and the *header*. The header starts at the entry point to the loop, before the body, whereas the continuation is executed after the body. Therefore, the first invocation of the header – because it is before the first `detach` – is outside of the parallel region, and so the dynamic trace is similar to the one seen before. The continuation and the (next) header are back-to-back in the trace, and they together form the induction variable update section, as shown in Figure 3.4(b). Figure 3.4(c) shows the parallel execution of this trace, just as before.

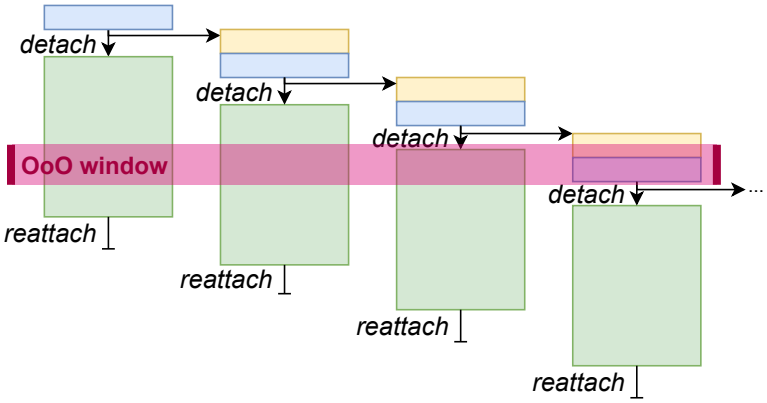
These hint instructions were inspired by Tapir [70]. As described in Section 2.8.1, Tapir embeds asymmetric parallelization directives into LLVM to support *explicit* (i.e. user-specified – Cilk or OpenMP) parallelism in the compiler, without breaking SSA form and most compiler analyses that rely on it (unlike symmetric fork-join directives). Here, instead of focusing on explicit parallelism, we utilize similar directives as hints for *implicit* (i.e. microarchitectural), speculative parallelism. Furthermore, unlike Tapir, the hints are inserted



(a) Loop structure and hints



(b) Sequential execution



(c) Parallel execution

Figure 3.4: Loops divided into *header*, *body* and *continuation* by hints. Static view (CFG), and dynamic (timings) view during sequential and parallel execution.

Listing 3.1: Example loop with different parts highlighted as follows: before/after region, header, body and continuation.

```
1     i = 0
2     j = 1
3     while (i < N)
4           i = i + 1
5           detach C
6           call foo(j)
7           reattach C
8           C:
9           j = j · 2
10    sync C
11    print "Done"
```

in a late middle-end pass (instead of using front-end directives), and then lowered all the way to machine code (with appropriate encoding changes). Each machine instruction carries the continuation block's address, which serves as the jump address for the newly-launched successor epoch, as well as a unique region ID, which can be used to match related hints together.

Listing 3.1 shows an example of the hints and the resulting structure applied to a simple example loop in pseudo-code. If the compiler emits all of these lines in the same order to the binary, then the update of the induction variable i will be in the header, and thus the condition check just before also ends up there. The call to the function `foo` is in the body, between the detach and reattach instructions, while the update of the variable j (also a loop-carried dependence) is in the continuation. After exiting from the loop, a sync instruction is executed before execution can proceed outside the loop.

3.2.3 Hint parallel semantics

A `detach` marks a potential fork point: execution can proceed in parallel from here (if the microarchitecture supports it). The current threadlet (see Section 3.5) continues executing the current epoch from the next instruction, and the next epoch can be launched in a new threadlet from the continuation address `C`, executing speculative work. The successor epoch inherits the register state of its predecessor upon `detach`. At this point, the current epoch has 'detached on region `C`', and it will ignore hints with a mismatched region ID. A repeated `detach C` hint would also be ignored, unless nested parallelism is exploited, as described in Section 3.2.5. That is, all hints except `reattach C` and `sync C` are ignored by the threadlet running this epoch.

`Reattach C` ends the epoch, as the next instruction in program order is the successor's first instruction. Thus the current threadlet can be recycled, ready to execute a future epoch. As soon as all effects of the current epoch have been committed (merged) to architectural

state, the successor's effects can begin merging. Once any of the successor's actions are observable in the memory system, it can no longer be squashed and restarted, as doing so would expose speculative parallelism architecturally.

Finally, `sync C` signifies that the current epoch is exiting the parallel region. This means there are no more instructions in the region, and so any successors that may have been spawned were spawned due to incorrect control speculation. In this case, all successors are squashed and discarded. Once the sync instruction is committed to architectural state (note that this is after the current threadlet becomes architectural), the current threadlet continues sequential execution of the code after the `sync C`.

3.2.4 Preserving sequential semantics

The sequential observable semantics of the original serial program must be strictly preserved, so that parallel behaviour is not exposed architecturally. The microarchitecture is responsible for maintaining the illusion of sequential execution. Since epochs are strictly ordered, there is a well-defined total order in which memory operations and side-effects should logically *appear* to happen. Thus, all threadlets execute *speculatively*, other than the oldest one, which is executing *architecturally*. For speculative threadlets, the microarchitecture transparently buffers all memory writes (and it pauses execution before side-effecting operations such as exceptions, barriers or input/output operations occur) until the threadlet becomes architectural (in sequential program order). Speculative threadlets can be cleanly discarded for any microarchitectural reason, such as true (read-after-write) dependence violations, exceptions, context switches or buffer overflows.

Each time a speculative threadlet becomes architectural, its buffered updates are merged with the global architectural state and are visible to the system. A speculative buffer implementation is given in Section 3.10.2, and an efficient implementation of the commit mechanism – which respects the memory consistency model – is described in Section 3.10.6.

Memory data flow and hazards

The architecture places no restrictions on any type of memory dependences (or hazards) between epochs. Instead, the microarchitecture handles each type of dependence as follows.

The multi-versioned, copy-on-write speculative buffers eliminate *write-after-read* hazards. Adding the in-order commit mechanism eliminates *write-after-write* hazards, as well as *read-after-read* hazards (if required by the architecture, such as on AArch64). *Read-after-write* (or true) dependences are supported, but they may lead to squashing.[4] The microarchitecture tracks memory read and write address sets for each epoch, and it squashes and re-executes any *conflicting* threadlets (i.e. those that have read any stale values due to a read-after-write hazard).

[4] In the base version – presented in this chapter – any true dependence between speculative epochs results in a squash.

Register data flow

As described in Section 3.2.1, the register values at the end of the header are copied to the next epoch when it is detached into a separate threadlet. Thus, register data flow from the update section (continuation or header) to successive sections does not lead to a hazard. On the other hand, data flow from the body to a future epoch causes a hazard, which needs to be monitored and resolved by means of squashing. However, there are three special cases, where we can avoid squashing, as described below.

Data flow out of the region There are no restrictions on register data flow out of the region. That is, an instruction after the end of the region *can* consume the last value (in program order) that was stored into a register inside the region. This is so that information about the parallel region can be discarded as soon as we exit the region at runtime (i.e. as soon as the `sync` instruction commits architecturally), and so that the scope of the compiler's analysis can be restricted to the parallel region itself. To support this, the microarchitecture records register read and write sets, and when a threadlet retires, its end state is sent to the successor. When the successor retires, or leaves the region, it merges the two end states as follows. Any registers written to by the successor are kept from the successor, whereas other registers are copied from the retiring thread's end state. The logic is detailed in Section 3.9.3.

Spill and fill We allow register dependences that arise purely due to register spill and fill patterns. This is where the compiler decides to spill (save) some registers to memory (typically onto the stack) due to register pressure. Although we handle more complex patterns, the motivating example for supporting this is spilling inside functions called from within the region. Any callee-saved registers that are clobbered inside a function are spilled, usually near the start of the function. This case is particularly important, as the spilling is done inside a function that is invoked from the loop body. Allowing this pattern removes the need for the compiler to inspect the code inside the function, thus keeping the parallelising transformation implementable without inter-procedural analysis. This is crucial for enabling arbitrary function calls, as the cross-function analysis is tricky to implement in a loop compiler pass, and the code may not be available due to it residing in a different compilation unit.

Thus, without the relaxation, the compiler would likely need to insert significant extra code to ensure no incidental register data flow occurs between the body and the continuation.[5]

Instead, the microarchitecture can monitor accesses that are consistent with the spill-fill metaphor, and patch up values in the spill slot in memory, if appropriate. The implementation is described in Section 3.11.3, where I also give a precise definition of the supported spill-fill pattern(s), using a state machine. The idea is that a register R is only observed by a simple store to a memory location M, then neither R nor M are read before being fully overwritten, except if M is loaded back into R by a simple load instruction. Multiple nested or successive spill-fill patterns on the same register are supported, where each part observes the same pattern.

[5] Specifically, all (potentially) modified registers (and all callee-saved registers, if there are function calls inside the body) would need to be saved at the start of the body and restored just before the end.

As soon as a deviation from the metaphor is found on a register that observed a conflict, the microarchitecture has to squash offending instructions and fix up all copies of the initial value. That is, the value of the register and any associated spill locations in memory need to be updated. Note however that the cost of this fix-up is not key, as correctly inserted hints do not result in register conflicts, and the spill locations are never read.

Unchanged value For the purposes of detecting register conflicts, we compare the read set of the successor with the *change set* of the current epoch (i.e. the set of registers whose architectural value differs between the detach and reattach points). This hides benign read-after-write conflicts, where the stale value read and the up-to-date value are identical. Tracking the change set is feasible – unlike for memory – due to the smaller number of registers, and it is more beneficial, as it permits common patterns, including spill and fill patterns[6] and function calls in the body. If function calls are present in the body, the invoked function (typically) decrements the stack pointer in order to allocate its stack frame. Before returning, the stack frame is then deallocated by resetting the stack pointer to the original value. Since the start and end of the body are in the same function, the stack pointer should be the same at these two points.

Exceptions, interrupts If an instruction in a speculative threadlet generates an exception, the threadlet is paused until it becomes architectural. When it does (or if the threadlet was already architectural), the exception can be taken. When an exception, trap or interrupt is taken (in the architectural threadlet), all speculative threadlets are squashed and recycled immediately, before taking the exception/interrupt. This means no speculative state needs to be saved and restored when entering/exiting kernel mode or performing context switches. This is a simple model, and improvements may be possible, although they may not be profitable. Speculation will naturally restart when the next detach instruction is encountered after restarting the program.

3.2.5 Nested regions

The compiler is permitted to generate nested parallel regions, such as a hint-annotated inner loop inside the parallel body. Supporting nested region placement is important, as the compiler may not have sufficient information to choose the most profitable level to parallelise, and it may be difficult or sub-optimal for the compiler to avoid such placements altogether. Specifically, if a function containing a candidate loop has multiple call sites, it is recursive, callable from other compilation units, or if the function itself calls functions in different compilation units, the compiler will struggle to eliminate nested regions without completely excluding a significant fraction of functions from hint insertion.

The microarchitecture can distinguish hints from different levels of the loop nest using the unique region ID field that is included in each hint. It may be the case that a region A in function f contains a recursive call to the same function, which could lead to recursive nesting of the same region. This case can be handled by not only comparing the region ID,

[6] This is necessary in conjunction with the spill and fill tracking described above, as we need to handle both sides of the access.

but also the value of the stack pointer register (or the function-call depth, which could be tracked by the microarchitecture). Since such cases were rarely encountered in the evaluation benchmarks, the prototype compiler simply disables parallelisation of regions that lead to recursive nesting.

The microarchitecture can choose how to handle nested regions. One option is nested parallelisation, maintaining ordering using concatenated epoch IDs in a similar way to Fractal [78]. However, since the number of threadlet contexts in a single processor pipeline will be low (e.g. 2 or 4), the prototype implementation simply chooses a single level to parallelise at runtime. A real implementation could use iteration size and conflict rate to pick the most profitable level (while the prototype imitates this using profiling).

3.2.6 Applicability to non-loop parallel regions

The same hints can be applied to patterns other than loops. In the simplest case, we can expand coverage to control-flow patterns that show up as loops in the CFG, even if they don't correspond to loops in the source code. An example is simple recursion with tail-call optimisation. These cases can be supported with no additional effort, so long as the compiler adds annotations sufficiently late in the pipeline (e.g. after the tail-call optimisation).

Asynchronous work can also be supported in general, although the prototype compiler does not support annotating such regions at the moment. Examples of this are a function call that does not produce a result, or whose result is not consumed for a while (because there is other work to do), and whose side effects likely do not affect code immediately after. This can be supported as shown in Figure 3.5.

The slight caveat is that this layout only yields a single pair of parallel epochs. To produce significant benefits, nested parallelism support can be added, perhaps explicitly

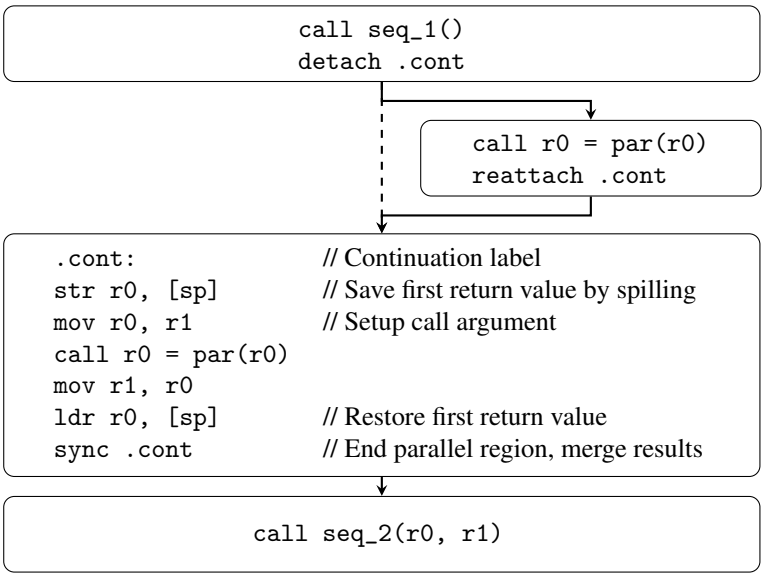


Figure 3.5: Asynchronous parallelism. The two calls to the function `par` are parallelised, and the return value from both are passed to `seq_2`, which runs in sequence after the parallel region.

tagging the hints as producing only a 1-bit epoch ID per level, to support more levels of nesting. A very common example that may benefit is divide-and-conquer algorithms, such as quick sort. Here, we find and sort the pivot into the correct position, then detach to sort the left-hand side (recursively) in parallel with sorting the right-hand side. The region (level of parallelism) ends with a sync before the return.

Due to the lack of compiler support at the time of writing, I do not explore these cases and additional optimisations in detail, instead focusing on loops, and only mention the possibility to complete the story.

3.3 Compiler

The compiler was developed by my colleagues, Utpal Bora and Akshay Bhosale. While I contributed some ideas, the approach described here should be credited to them. I describe it for completeness, as its feasibility is important for supporting the hypothesis.

We developed a compiler pass in LLVM [45] to perform hint insertion. The pass is based on OpenCilk [69] and Tapir [70], where we implemented middle-end hint insertion, modified the fork-join hints to match the hint semantics described in Section 3.2.3, and added a back-end version of each hint. Candidate loops can be selected using hand-inserted `#pragma` annotations, or the compiler can transform all loops. Future work could implement heuristics to choose candidates automatically.

Code is first compiled using standard optimization level `-O3`, which includes vectorization. Then the compiler processes each loop in turn, identifying possible `detach-reattach` pairs by executing algorithm 1 followed by algorithm 2. The condition on line 4 of algorithm 1 identifies blocks that will execute *exactly once* per loop iteration. Blocks in any inner loops are excluded, as they could execute more than once. Blocks *BB* that may only execute in some iterations are excluded by inspecting the *maxID* variable. For example, in Figure 3.6, `bb2` may not execute in each iteration, as the edge `bb1` $\rightarrow$ `bb3` skips it. This is detected as $maxID = \text{rpo\_no}(\texttt{bb3}) = 3$ when `bb2` is visited, but $\text{rpo\_no}(\texttt{bb2}) = 2$, so the condition on line 4 fails.

Algorithm 1 Finding reattach positions.

Require: Loop: L
Ensure: Reattach Points: RP

1: **function** REATTACHPOINTS(Loop L)
2: $maxID \leftarrow 0$
3: **for all** $BB \in \text{reverse\_post\_order}(L)$
4: **if** $\text{rpo\_no}(BB) \geq maxID \wedge BB \notin \text{inner\_loop}(L)$
5: RP.add(BB.instructions())
6: **for all** $succ \in \text{successor}(BB)$
7: $maxID \leftarrow \max(maxID, \text{rpo\_no}(succ))$
8: **return** RP

Algorithm 2 Finding detach positions.

Require: Loop: L
Require: Reattach Points: RP
Ensure: Detach Points $\langle RP; \text{detaches}(RP) \rangle$

```
1:  function DetachPoints(L, RP)
2:      for all R ∈ RP
3:          DP ← {R}
4:          IllegalDataFlow ← False
5:          for all BB ∈ post_order(L)
6:              if BB ∉ RP
7:                  continue    // Illegal control flow
8:              if po_no(BB) < po_no(R.block())
9:                  continue    // Below reattach
10:             for all I ∈ reverse(BB)
11:                 if BB = R.block() ∧ I ≥ R
12:                     continue    // Below reattach
13:                 if I ∉ live(R)
14:                     DP.add(I)
15:                 else
16:                     IllegalDataFlow ← True
17:                     break
18:             if IllegalDataFlow
19:                 break
20:         detaches(R) ← DP
21:     return ⟨RP; detaches(RP)⟩
```

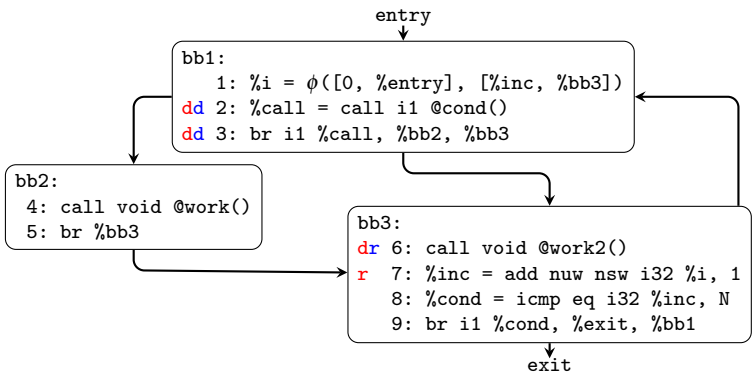


Figure 3.6: Hint placement using algorithms 1 and 2. Two reattach points (r) have non-empty bodies and multiple detach points (d): $7 \mapsto \{2,3,6,7\}$ and $6 \mapsto \{2,3,6\}$. The result is a detach at instruction 2, reattach at 7 and a sync at `exit`.

Credits: Utpal Bora.

To obtain the set of possible detach points for each reattach point R, algorithm 2 iterates through all blocks in post-order, starting from R (lines 5–9).

For each block BB in the loop, the instructions I are investigated in reverse order for data-flow from the body to future iterations (lines 10–17). Values that are produced inside the body and are (still) live at the reattach point would create a data-flow dependence between iterations. Therefore, if such a producer instruction is encountered, the body cannot be expanded further, and we stop processing the reattach point (lines 15–19). For the example in Figure 3.6, for $R = 8$, when investigating $I = 7$, we find that the produced variable `%inc` is live at R, therefore *IllegalDataFlow* is set, and $detaches(8) = \{8\}$ is not expanded further.

Note that due to being in SSA form, it is sufficient to investigate blocks in PR for variable definitions, as a definition in other blocks will create a ϕ-node in PR (unless the value is no longer live). We also never insert reattaches or detaches *above* any ϕ-nodes in a basic block (not shown here), as doing so would result in illegal SSA form.

After obtaining candidate `detach-reattach` pairs, the compiler picks the body with the largest number of static (IR) instructions along the longest linear path. This is a simple heuristic for maximizing the number of dynamic instructions at run-time. Loop-carried memory dependences may hurt performance, but minimizing these is left as future work. Furthermore, while this heuristic is correct for *overlapping* bodies[7], it may choose sub-optimally between *non-overlapping* ones. For example, a body with a single call instruction could have more dynamic instructions than a long sequence of instructions. Nevertheless, we found that this heuristic works well in practice.

After picking the boundaries of the body, the compiler inserts Tapir IR instructions. These are then lowered to hint machine instructions in the AArch64 compiler backend.

Limitations

The above approach is able to transform thousands of loops in the SPEC CPU2006 [33] benchmark suite, and other programs correctly, with only a handful of failures (e.g. the *gcc* benchmark in the SPEC suite), and a small number of loops skipped for technical reasons (e.g. if LLVM cannot transform them into a canonical form). On the other hand, a number of loops cannot be transformed due to specific patterns of dependences, and we believe that many loops could obtain a performance boost from certain optimisations.

The two main artificial limitations at the time of writing are the lack of code motion, and requiring *detach* and *reattach* hints to each be executed exactly once (in order) in every loop iteration (except the last one, in case of an early exit from the body).

In terms of code motion, there are two obvious approaches to improve the algorithm. We could try and move as many instructions as possible (without changing program semantics) into the body, in order to increase its size, or we could move problematic (conflicting) instructions (and their dependences) to the start and end of the loop iteration, before choosing the body to annotate, in the hopes that a larger body can be selected due to the transformation.

[7] Using the fact that blocks in RP each execute exactly once per iteration, in the same order, as well as the contiguous property of SSA liveness, we can show that for any pair of overlapping bodies A and B, $A \cup B$ is a valid body.

Although there is some overlap in the effects, two approaches could also potentially be applied in tandem.

As for the placement of *detach* and *reattach* hints, the only restrictions the hint semantics impose are that they need to be control equivalent within the loop iteration (not considering loop exits that are annotated by *sync*), and *detach* needs to come before *reattach* in program order. We have observed several loops, where very few (typically only two) basic blocks are executed exactly once per iteration, thus the current compiler only considers a very limited set of possible hint placements. Often this is because most of the loop body is inside a conditional branch.[8] In these cases, putting the *detach* and *reattach* inside the conditional can potentially yield a larger valid body.

Both of these limitations can lead to loops where profitable hint insertion is not possible with the current compiler, or the resulting speedups may be less than they would be with more optimal hint insertion.

3.4 Microarchitecture overview

This section gives an overview of the microarchitectural changes necessary to add support for in-core, hint-based, microarchitectural speculation.

Baseline Pipeline

Figure 3.7 shows a baseline (unmodified) out-of-order processor pipeline for reference. The diagram is not meant to represent a specific design. Rather, I give it here to map out the logical components mentioned later in the thesis, and where they fit into the pipeline's workflow. The exact set of structures, exact layouts and naming conventions vary between CPU vendors and microarchitectures, but this does not materially impact our design in most cases.

The pipeline consists of the front end (instruction fetch, decode, rename and dispatch), and the back end (out-of-order engine, instruction commit).

Bytes of machine code enter the pipeline from the level-one instruction cache, as they are fetched into the fetch buffer. The fetch buffer contains an aligned portion of the relevant cache line. The instruction fetch stage picks and separates instructions from the data in the buffer, predicts the next program counter address, and requests the next fetch buffer segment (containing the next PC) from the instruction cache (if necessary). Fetched instructions are put into the fetch queue, which has space for a few cycle's worth of fetched instructions. This allows the fetch stage to slightly run ahead of the rest of the pipeline, which helps to decouple delays between fetch and the rest of the stages. This is helpful, as fetch delays occur on a per-cacheline granularity, whereas the rest of the pipeline works on a per-instruction basis.

Decode processes instructions in order, unpacking the opcode and operand fields. Register rename also observes the instruction stream in order. It looks up the physical register number of each operand in the rename table, and allocates new registers for any outputs.

[8] In C/C++, this can also be expressed by an *if-condition-continue* statement, but it results in equivalent LLVM IR and machine code.

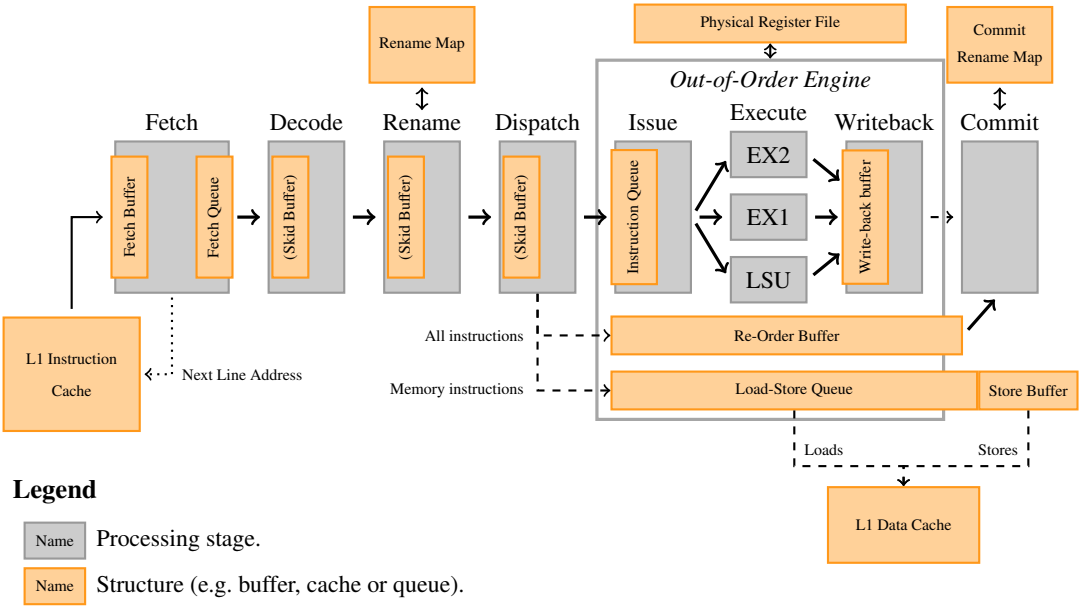


Figure 3.7: Overview of an unmodified baseline out-of-order pipeline, for reference. Gray boxes represent stages of processing, while gold boxes are storage structures (caches, queues, buffers).

Dispatch ensures there is sufficient space in each of the required back-end structures (i.e. instruction queue, re-order buffer, load-store queue), and inserts the instructions if possible (in order). All of the front-end stages have an upper limit on the number of instructions they can process, known as the (front-end) *pipeline width*.

It may be the case that the next stage is unable to take instructions for some reason, forcing the current front-end stage to block and retain the current set of instructions for an extra cycle. For example, one of the back-end structures may be full, causing dispatch to block, or rename may be unable to find a sufficient number of available physical registers to allocate the outputs of an instruction. In these cases, the previous stage will be unaware of the blockage, and it will still send a new set of instructions in the next cycle. Since the blocking stage still holds the set of instructions from the previous cycle, these new instructions are saved into a small structure called the *skid buffer*. The previous stage will now block, and so the skid buffer only needs to be able to hold one cycle's worth of instructions. Once the next stage can process instructions again, it will process instructions in the skid buffer, and instruct the previous stage to unblock, and continue providing new instructions. Although its implementation varies between microarchitectures, this blocking mechanism is called *back pressure*, and it is fundamental in order to allow the pipeline to function in the presence of bottlenecks. If a stage is bottlenecked, pre-processed instructions from previous stages are held and buffered, and previous stages eventually block before the buffer over-fills.

The out-of-order engine starts with the issue stage, which chooses instructions to issue to the execute pipes. Instructions are chosen if their inputs are ready (based on the register *scoreboard*), and there is a free execute pipe that is capable of executing that type of instruction. There are different types of execute pipes (e.g. simple/complex integer arithmetics, floating point and vector), and there may be multiple copies of each type. The load-store unit

is the pipe that processes memory instructions. Execution starts by reading out the inputs from the register files, then performing the operation itself (e.g. arithmetics, memory access). Once instructions leave the execute pipes, they are inserted into the write-back buffer. The writeback stage pops as many instructions as possible (up to the *writeback width*) from this buffer, stores their outputs into the register files, it marks those registers as ready on the scoreboard, and it marks the instruction as ready to commit in the re-order buffer (ROB).

The commit stage takes ready instructions from the front of the ROB (up to the *commit width*), and deallocates their resources. All committing instructions are deleted from the ROB, and the commit rename map is updated based on their outputs. For example, if an instruction produced the architectural register `x10` into physical register `p130`, then the commit rename map is updated with the mapping `x10` $\mapsto$ `p130`. If there are no more in-flight consumers for a given physical register (and it is not in the commit rename map), then that register is recycled. Further to this, commit also frees up the load-store queue entries associated with committing instructions, and it inserts any committing stores into the store-buffer, from where they can be written back to the memory system in the background.

If faults or mispredictions occur, the stage that detected them sends out a squash signal with the associated instruction sequence number. All stages then *squash* younger in-flight instructions, either by discarding them or by turning them into NOPs (operations without observable effects), and fetch restarts from the appropriate place.

Modified Pipeline

Figure 3.8 shows the pipeline with added support for hint-based speculative multithreading.

The CPU pipeline is augmented with lightweight threadlet contexts. These are similar to thread contexts in simultaneous multithreading (SMT). Each context has its own control state (e.g. ROB and LSQ slice, fetch buffer, fetch queue, program counter, rename maps) and architectural registers, and every instruction flowing through the pipeline is tagged with a context ID. Thus, threadlets are isolated from each other, as they cannot read each other's registers, and pipeline squashes (due to mispredicts or faults) only affect one threadlet. On the other hand, unlike SMT threads, threadlets are not visible to the operating system or programmer, and they run code from the same process, mapped by the microarchitecture using hints described in Section 3.2. The microarchitecture provides the illusion of sequential execution by hiding writes from younger threadlets from reads from older threadlets, and monitoring ordering violations.

The speculative state buffer (SSB) is the heart of the microarchitecture. It buffers speculative writes, and hides them from older threadlets, and it serves the correct data to loads coming from speculative threadlets. For architectural accesses (from the oldest threadlet), the SSB is bypassed (but it may be updated in the background), thus the SSB does not increase access latency for architectural accesses (and during sequential regions). For accesses from speculative threadlets, values in the SSB are looked up in parallel with the data-cache lookup, and the return value is patched together from the value received from the data-cache and different epochs (not younger than the load instruction) in the SSB to obtain the most up-to-date copy.

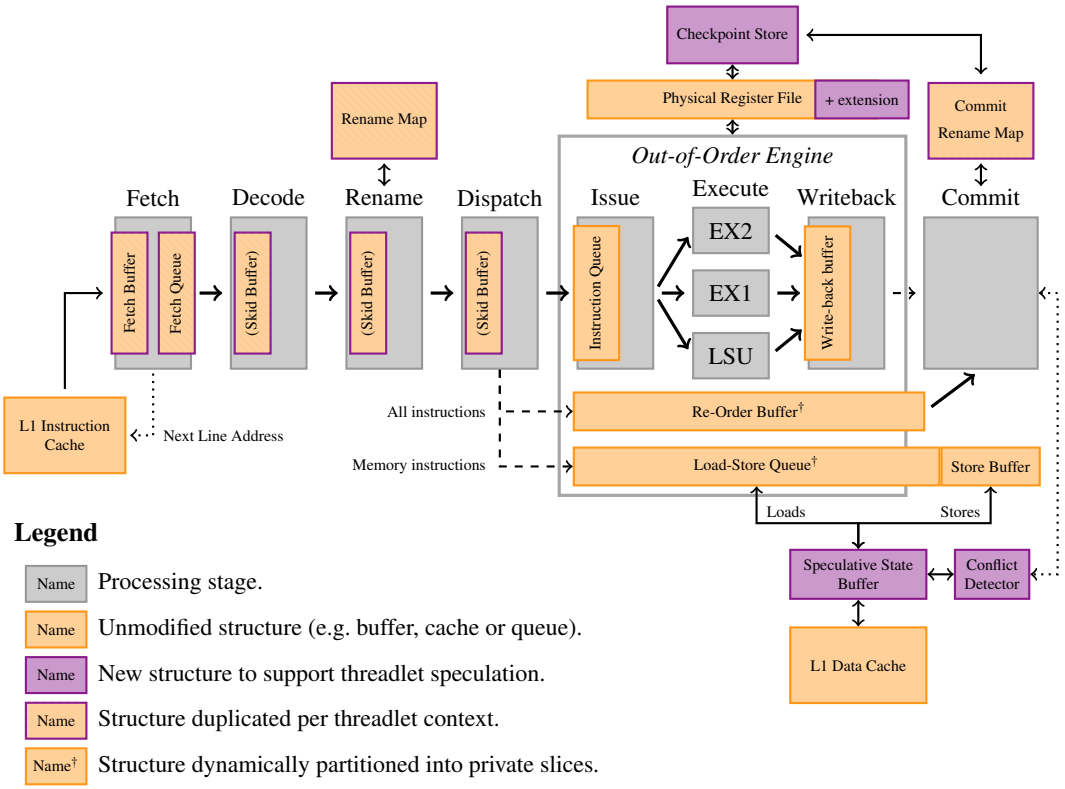


Figure 3.8: Overview of the out-of-order pipeline with support for threadlet speculation.

The conflict detector's job is to work out if misspeculation has occurred and a speculative threadlet has observed stale data. If so, it triggers corrective action (squashing of offending threadlets).

The checkpoint store keeps the starting (architectural) register state of each threadlet to facilitate restarting after a squash, and to facilitate conflict checking for register values.

In addition to these new structures, most pipeline stages are modified in some way to facilitate threadlet-based parallelism. The fetch queues are duplicated in order to better simulate powerful modern front-end pipelines with features such as fetch-directed prefetching [36]. State-of-the-art baselines may not require this duplication. Other units are shared between the threadlets. The physical register file is extended slightly to hold the architectural register states of all threadlets.

The next sections describe how the different parts operate in more detail.

3.5 Threadlets

A threadlet is a lightweight execution context. Similarly to fully-fledged SMT threads (or contexts), each threadlet has its own (user-space) architectural register state and program counter, as well as dedicated microarchitectural control state (e.g. in-flight instructions, start and end of instruction window slice). On the other hand, instructions share (most) pipeline resources, and their effects are directed to the correct threadlet using a context ID tag carried by each operation and each entry in pipeline structures (such as buffers and queues).

Active threadlets in the same core always correspond to the same process, which means they share the same virtual address space (and thus TLB and L1 cache entries) and system registers. Furthermore, protection between threadlets is not a priority. The reordering of instructions between threadlets can be seen as an extension of out-of-order execution, which is already permitted under threat models. Mitigations for exploits like Spectre [42] should apply similarly to simple out-of-order execution. A compiler implementation should also provide an option to turn off hint insertion for security-critical portions of code, for example via source-code annotations.

Lastly, the threadlets have a well-defined total order between them (based on their epoch IDs), which can also be used to prioritise older, less-speculative threadlets. As described in Section 3.2.4, all but one of the threadlets run in speculative mode, and memory values are inherited in epoch order. Thus, the cache system must keep private copies separate between the threadlets, and work out the final value after each threadlet commits.

3.6 Threadlet lifecycle

This section describes the lifecycle of threadlets contexts in the microarchitecture. A summary is shown in Figure 3.9.

There are a fixed number of threadlet contexts (e.g. 4), onto which the microarchitecture can map program epochs. During sequential regions, such as at the start of the program, one of these threadlets is in the *running* state, executing the program in 'architectural' (non-speculative) mode, just like the baseline system would. The remaining threadlets start in the *idle* state.

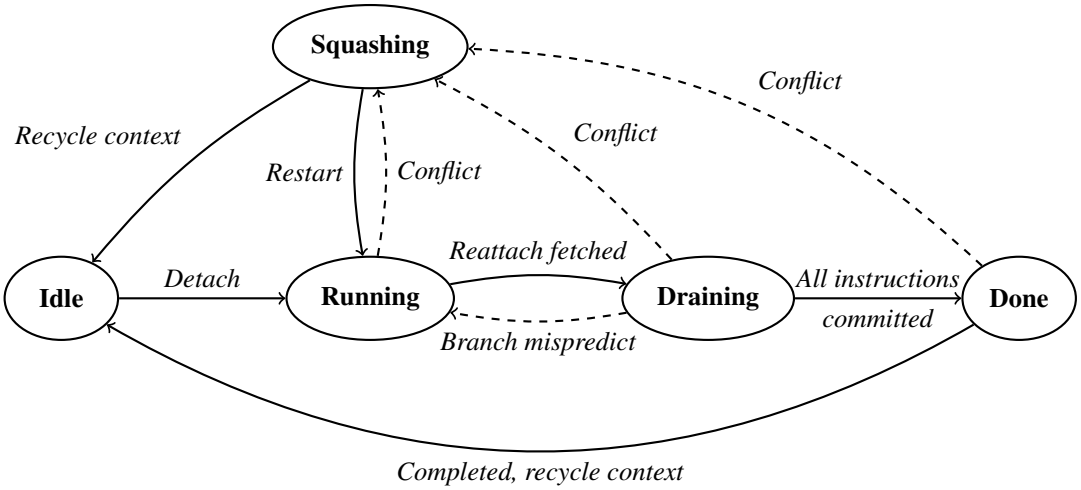


Figure 3.9: The lifecycle of threadlet contexts

3.6.1 Architectural and speculative execution modes

Threadlet contexts that are not *idle* are considered *active*. On top of the classification based on lifecycle stages, threadlets may be architectural or speculative. Threadlets start as speculative, and the transition to architectural execution may happen in the any of the three main stages (that is, *running*, *draining* and *done*), as the transition is triggered when the predecessor threadlet sends a commit request. The transition itself takes some time, as described in Section 3.10.6, and when it is finished, all prior completed stores from the threadlet are architecturally visible in the memory system.

3.6.2 Detaching a new threadlet

When a threadlet executes a matching[9] detach instruction, it attempts to detach a continuation. To do this, it searches for an *idle* threadlet context to launch the continuation epoch on.

If no *idle* threadlets are found, the detach is not taken (ignored). Otherwise the *idle* threadlet is chosen as the successor, its PC and architectural register state is set up based on the state of the detaching threadlet and the detach target address, and the newly started threadlet transitions to the *running* state.

3.6.3 Front-end and back-end activity

A threadlet in the '*running*' state is both fetching new instructions in the front end and executing and committing them in the back end. Once a running threadlet that has previously detached a successor fetches a (matching) reattach instruction (or just before it fetches from the continuation address)[10], we know that instruction fetch has caught up to the successor's starting point, therefore fetching in this threadlet stops and the threadlet transitions to the *draining* state. Just like in the *running* state, the back end continues to execute and commit instructions, but the front end will not fetch any new instructions.

Since the reattach is the youngest instruction in the out-of-order window, any branch mispredicts that are detected in a *draining* threadlet will cause the reattach instruction to be squashed. If this happens, the threadlet returns to the *running* state, until it fetches the matching reattach instruction along the new (more correct) code path. The pair of transitions can happen multiple times, until all branches are resolved correctly (making some forward progress each time).

3.6.4 Recycling completed threadlets

Once all instructions in the epoch have committed to the threadlet, the threadlet is finished with execution and commit, so it transitions to the *done* state. Here, it waits until three things successfully happen. Firstly, all stores need to write back from the store buffer to the SSB, in order that they can become architecturally visible later. Secondly, the threadlet needs to become the oldest in the processor and successfully commit its effects architecturally. Before

[9] That is, a detach whose region ID matches the ID of the current active region, or any detach when not inside any regions.

[10] We can either find out using a PC check in instruction fetch, or by inspecting the instruction in decode and squashing any further instructions that may have been fetched.

this happens, a conflict can still occur if an older threadlet writes to a memory location that the current one has read. Finally, it needs to send a commit request, initiating a commit of its successor to architectural state. If the successor is already in the *done* state, this will be a full commit, if it is still *running* or *draining*, then this will be a partial commit. In both cases, all the existing speculative state of the committing threadlet is written back. However, full commit means that this completes the execution of the epoch, whereas in the case of partial commit, the epoch still has work to do, which it can finish in architectural mode, after the commit. If the committing threadlet is *squashing* (and about to restart), we wait for the squash to happen, and the successor will simply restart in architectural mode.

Once all three of these things happen, the threadlet can complete and retire. All threadlet metadata is discarded, and the threadlet context is recycled into the *idle* state to be used to execute a different epoch in the future.

3.6.5 Failure and squashing

Until the transition to architectural mode finishes, the threadlet is speculative, and thus may be squashed. A squash will be triggered if a true (read-after-write) memory conflict is detected between the threadlet and a predecessor, or if a matching sync instruction is executed by the predecessor, and the microarchitecture is free to squash at any point it decides to (e.g. upon a context switch, system call or buffer overflow). If a squash is triggered, the threadlet moves into the *squashing* state. Once all in-flight instructions and memory operations are squashed, there are two possibilities for each threadlet. It is either recycled or restarted. If we squashed multiple threadlets at once, all of them except the oldest one are recycled into the *idle* state. The oldest squashed threadlet may also be recycled, but it may be restarted instead. If it is restarted, the starting checkpoint is read out from the checkpoint store, and its content is used to populate the architectural registers and the program counter of the restarted threadlet, similarly to a detach. A restart will typically be performed after a memory conflict triggers a squash (as the work of the squashed threadlet still needs to happen), but it will not be done after a sync instruction (as the threadlet was only ever started due to incorrect control speculation).

3.7 Resource allocation and priorities

We need to allocate pipeline resources between the different threadlets in the pipeline. The constraints and trade-offs for resource allocation are different from allocation in traditional SMT. SMT threads are independent, and typically of equal priority (in the microarchitecture). On the other hand, threadlets have unresolved dependences and a total order between them. We can intuitively expect prioritising older threadlets to be beneficial, for the reasons described below.

The main reason is that older threadlets are less speculative, both in terms of control speculation and dependence speculation. Committed instructions in the architectural threadlet are final, whereas those in speculative threadlets can be cancelled if an older threadlet exits the region, if a true read-after-write data dependence violation is discovered, or if speculation is abandoned for some other reason (e.g. context switch, speculative buffer full). While

in-flight instructions in any threadlet may be dependent on branch outcomes and exceptions from preceding in-flight instructions in the same threadlet, those in speculative threadlets are additionally dependent on inter-threadlet control and data dependence speculation. It is beneficial to prioritise less speculative work, as it is more likely to help the application make progress.

If instructions dispatched to a bottleneck resource have a (geometric) mean probability p of correctness (i.e. probability $1 - p$ of squash), then only a fraction p of slots are useful, thus misprediction causes a factor of p slowdown. By choosing less speculative instructions, with a probability $p' > p$ of correctness, we can obtain a speedup of $\frac{p'}{p}$.

Additionally, prioritising older threadlets reduces reordering, which – in turn – reduces the probability of conflicts between threadlets. In particular, as we will see in Section 3.11, if an older epoch retires before a younger epoch starts, then we do not need to check for conflicts between these two epochs. This means that we would prefer epochs to retire as early as possible, in order to reduce the number of speculative epochs that run in parallel with them.

Reducing reordering of operations from different epochs can also remove illegal re-orderings, even if the epochs still overlap. Unfortunately, as discussed in Section 3.10.2, the younger operation may still observe a stale value in this case, and the conflict detector will still find a conflict (see Section 3.11). However, these assumptions will be re-visited in Chapter 4.

Finally, committing epochs earlier is also beneficial, for two reasons. Firstly, speculative state can be written back to the memory system at epoch commit, and so it can be evicted from the SSB. Thus, committing early reduces the required size of this buffer. Secondly, since we perform conflict checking at commit time (see Section 3.11), an earlier commit also leads to finding conflicts earlier. In turn, this means that any conflict-based squashes happen earlier too. An earlier squash means that the epoch can restart – and execute the correct stream of instructions – sooner, which saves time upon squashing. Thus, committing epochs early reduces the conflict penalty, and leads to better performance on average.

Therefore, the system described in this chapter gives strict priority to older threadlets, whenever it picks the next instruction to be processed at any stage in the pipeline, whenever it chooses which instruction(s) to decode, rename, dispatch, issue, execute, write back or commit (when there is limited bandwidth). That is, each stage first tries to pick as many instructions as possible from the oldest epoch, before trying instructions from the second-oldest one, and so on.

Since there are no limits on the number of instructions from each threadlet, this can lead to the oldest threadlet using all the resources, and the speculative threadlets starving. However, there are a couple of important things to note here.

Firstly, while certain threadlets may proceed very slowly at first, permanent starvation or deadlock cannot occur, due to the strict ordering of epochs. Since the older threadlets can always make progress, they eventually finish, and give up their resources, allowing the younger threadlets (or at least the oldest one of them, which is now architectural) to claim resources and make progress.

Secondly, it may seem counter-intuitive to the aims of our research to reduce parallelism by favouring the oldest threadlet. After all, if the speculative threadlets never make progress,

then the system does not exploit any threadlet-based parallelism, and so the whole concept of speculative threadlets is pointless. However, this does not contradict the arguments above for prioritising the oldest threadlet. In the end, the processor pipeline has a finite set of resources, and the best we can do is optimise how these resources are used. In case the oldest threadlet can always usefully utilise all of the resources, then the system is already running at its peak throughput. Then, the only way to increase performance is to add pipeline resources, and thus change the baseline. This argument hinges on the fact that the architectural threadlet only claims resources that are *useful* to it, in the sense that they help to speed up progress.

Finally, note that priority inversion may still occur, as we use non-preemptive resource allocation for pipeline structures (other than SSB space). For example, an instruction from a younger threadlet may be holding up a slot in the load-store queue until its memory operation is serviced, even if an instruction from an older threadlet would be ready to go. It is hard to avoid such priority inversion, as deciding when preemption is profitable is complex due to work already performed.

3.8 Frontend

The frontend pipeline is the in-order section at the front of the out-of-order CPU pipeline. Instructions are fetched, decoded, renamed, and dispatched to the backend here. Each stage is limited to a set number of slots, each of which can hold one instruction per cycle. There are multiple possible designs when implementing response to back-pressure from the backend. Here, I use one based on having one skid buffer in each stage, capable of holding one cycle's worth of instructions. Each stage blocks when it cannot send all of its current instructions to the next stage, causing any incoming instructions to be inserted into the skid buffer. The stage unblocks when the skid buffer has been drained, instructing the previous stage to unblock and send more instructions in the next cycle.

3.8.1 Stall logic with threadlets

If we want to allow some but not all threadlets to block when insufficient resources are available, then we need to change the stall logic.

In the simplest case, we can duplicate the fetch queues and skid buffers for each threadlet context in each pipeline stage. Alternatively, the skid buffer can be dynamically partitioned, monitoring the number of total entries. If the microarchitecture is about to run out of skid buffer spaces, buffered instructions from younger threadlets can simply be squashed and re-fetched, even without squashing the threadlet itself.

In either case, these structures are small – containing information about two to three cycles' worth of instructions – therefore their impact on the design of the chip is limited. The prototype opts for duplicating skid buffers.

3.8.2 Instruction fetch

The pipeline starts with the instruction fetch stage. Here, branch prediction is used to predict the addresses of the next instructions. The cache line (or set number of lines) containing the

next predicted instructions is fetched from the instruction cache into the fetch buffer. From here, instructions are picked based on predicted branch addresses and targets.

Modern processors rely on fetch-directed prefetch (FDP) [36] to fetch many cache lines in advance. With this technique and other optimisations (e.g. instruction-cache prefetching), modern workloads with reasonable branch prediction accuracies tend to be backend-bound, meaning that there is under-utilisation in the fetch stage. To benefit from this, we can duplicate the fetch queues, and fetch buffer, in order to make it efficient to switch between which threadlet is fetching, and to have a pool of instructions ready from each threadlet for the decode stage (if there is sufficient bandwidth available).

This under-utilisation is likely sufficient to fetch from multiple threadlets without increasing the number of instruction-cache ports, although I could not test this due to the shortcomings of gem5's frontend, which does not implement FDP. Instead, I model the under-utilisation by increasing the number of instruction-cache ports in the prototype.

Instruction cache

The loops targeted by in-core speculative parallelisation need to be small enough to keep the amount of speculative state low. This means that their code – or at least the hot paths – typically fit into the first-level instruction cache. Adding more threadlets does not change this, and therefore instruction cache size is not a (new) bottleneck, which keeps the number of first-level instruction cache misses virtually unchanged (and negligible compared to data cache misses).

Traffic to the first-level instruction cache will increase somewhat by the introduction of threadlets. Adding cache ports can help handle the increased throughput requirement. Due to the read-only nature of the instruction-cache, the complexity of adding some ports is not prohibitive. Furthermore, due to the expected under-utilisation here, in real systems the extra port requirements may be limited, although it was not possible to test this in gem5.

Branch prediction

Similarly, the impact of parallelisation on the amount of prediction metadata is nominal, since the executed hot loop remains unchanged, as long as threadlets share the main predictor tables. Keeping global and local histories is challenging. Maintaining an approximate global history is easier, as we can simply do it by threadlet, and copy state on detach. This will not be fully accurate, as it will usually only contain the history accumulated in the headers and not the bodies (as the body was skipped by the detach). Keeping and merging local history (per branch) is very tricky, since the direction of branches committed inside a speculative threadlet may still change if the threadlet is squashed and replayed, and branches in different threadlets may commit out of order order, thus complicating training.

Handling this is an interesting design space. We may buffer resolved conditional branches (address and branch decision) per threadlet, and apply updates to the branch predictors sequentially when the threadlet commits, but doing so still leaves history incorrect inside speculative threadlets. We could duplicate branch predictors per threadlet, but doing so is likely not beneficial for two main reasons: firstly, it increases the storage requirement, and secondly – due to the relatively short-lived nature of our threadlets – warming up the

predictor for each context is likely a large cost. Instead, accepting that prediction is slightly worse for speculative threadlets is likely a more feasible option.

The simplest approach (used in the prototype) is to use – both query and update – a single, shared predictor as normal, and accept the increase in branch misprediction rates as a result. Although experimental data do not show a large jump in misprediction rates, due to the resilience of the complex LTAGE [72] predictor used, this is an area for future work, as microarchitectural performance can be sensitive to even small changes in misprediction rates in practice.

3.8.3 Decode, register rename and dispatch

The instruction decode stage is straightforward. Up to one pipeline width's worth of instructions come in from instruction fetch per cycle, and they are decoded and forwarded to the register rename stage.

Register renaming is performed separately for each threadlet. A separate rename map is maintained for each threadlet, and instructions look up and update information from the appropriate map (or use values produced by a preceding instruction, which is from the same threadlet and got renamed in the same cycle). Physical register files may be shared between threadlets, or left separate. Here, I model a large shared register file, as this allows all registers to be used in sequential mode.

When a new threadlet is detached, the (commit) register map of the predecessor is copied to the new threadlet. Since detach is performed at commit time, we know that all of these registers have already been written back, and therefore their values are now final and constant. This means that both threadlets can run as usual, and they will not overwrite any physical registers that are shared with another threadlet. The register freeing logic (including free lists) is modified to ensure that such shared mappings are released (only) when no users remain. That is, when all the threadlets using the mapping have moved to a newly created mapping (after overwriting the architectural register), or they have been squashed and recycled. This can be achieved by adding a usage bit-mask with one entry per threadlet context per physical register.

Dispatching instructions inserts them into the instruction queue (IQ), load-store queue (LSQ) and reorder buffer (ROB). From here, they will be issued, executed and written back out of order by the backend, once their inputs are ready and there are sufficient resources available. Like all previous stages, dispatch prioritises older threadlets.

3.9 Backend

The backend pipeline consists of an out-of-order portion, including the instruction window and functional units to enable out-of-order execution, and the in-order commit stage. To control out-of-order execution and squashing, all instructions are inserted into the re-order buffer (ROB) slice of the corresponding threadlet, and memory instructions are inserted into the load-store queue (LSQ) slice, which controls forwarding.

3.9.1 Dynamically slicing the ROB and LSQ

The ROB and the LSQ need to be divided into independent slices when multiple threadlets are executing in parallel, since instructions from different threadlets will be inserted and removed (committed or squashed) independently, at different times. This way, the slices can emulate fully independent ROBs (and LSQs) per threadlet, without paying the cost of duplicating these – relatively large – structures.

We can subdivide these units dynamically, using a linked-list structure, which can be done with a low overhead, and it has been implemented and deployed in commercial simultaneous multithreading systems, including IBM Power5 and Power8, and likely others [18]. The idea is to subdivide the ROB's memory into chunks, each containing a handful of instructions. The chunks are then chained together into a linked list for each slice. To allow the distribution of resources between the slices to change, the chunks can be allocated on-demand, from a shared free list. This allows for allocation at the granularity of chunks, with a maximum of two chunks' worth of entries lost to fragmentation per threadlet (but one of the two still being available to that threadlet).

The load-store queue can be sliced similarly. In addition to being able to allocate resources dynamically, slicing is also important in order to ensure that values are only forwarded from stores within the same threadlet. If the LSQ implements parallel lookup, entries can be returned or not based on checking a threadlet tag associated with each LSQ chunk. Note that overall lookup bandwidth is not modified, as the number of load-store pipes remains the same between the baseline and the parallel core.

3.9.2 Out of order pipeline (issue-execute-writeback)

The instruction issue stage schedules instructions whose inputs are ready from the instruction queue to a suitable (free) functional unit. Functional units (such as simple or complex ALUs, floating point units, etc.) execute the instructions. This entails computing the output value from the inputs, either via arithmetic logic, or by sending a load request to the memory system. Store instructions perform address calculation and store their values into the load-store queue. From here, younger loads reading the same location can access the store value, but the rest of the system (other cores and peripherals) will only observe the stored value once it has been committed and written back.

Then, executed instructions write back their register results to the physical register file in the writeback stage, and they are marked as ready to commit.

Since register rename already ensures isolation between instructions from different threadlets, just as far-away instructions inside the out-of-order window would be separated normally, these stages do not require significant modifications.

The exception to this is memory load instructions – we must make sure they get the correct memory value. For this, we add threadlet tags to each entry in the load-store queue (and the store buffer), and ensure that any forwarding only occurs from the preceding instructions from the *same threadlet*. Together with modifications to the memory system (as summarised in Section 3.4 and detailed in Section 3.10), these modifications ensure the correct handling of memory instructions.

3.9.3 Instruction commit

At the end of the pipeline, the commit stage processes instructions within each threadlet in order. Starting with the oldest threadlet, instructions that are marked ready at the front of the reorder buffer are committed one-by-one, until either the next instruction is not ready to commit yet, or the commit bandwidth is reached. If any bandwidth remains, the next threadlet is processed similarly, and so on. Committing frees up resources associated with the threadlet (load-store queue slot, reorder buffer entry) and updates the commit rename map, thus updating the architectural register state of the threadlet.

Committed memory store instructions are also (immediately) inserted into the store buffer, which holds them until they are written back to the memory system (or the speculative state buffer in our case) in the background.

The instruction commit state is the natural place for performing many parallelisation-specific tasks, because this stage encounters instructions in-order, and it observes up-to-date architectural state (that is, the register writes by preceding instructions are visible in the architectural register file, and the commit program counter is know too). Furthermore, committed instructions cannot be squashed due to branch mispredictions.

Detach instructions

As discussed in Section 3.6, we first determine whether the detach instruction matches the current region ID (if we are currently inside a region). If it does not, the detach is treated as a NOP (i.e. it has no effect). If it does, we launch a new threadlet (if a free context is avaialble). The architectural register state and program counter are copied into the checkpoint store, and into the successor threadlet context, which then starts running the continuation epoch.

If the current region ID is empty when the detach is executed, then the detach has entered a new region. In this case, the current region ID is set up, and we may reset some statistics to count events on a region-by-region basis.

Reattach instructions

Matching reattach instructions are processed twice in commit. The first time is when they reach the front of the ROB in their threadlet context. At this point, all prior instructions have committed. The reattach signals back to the memory pipe to issue and execute the reattach, which causes a commit request to be sent (once the instruction executes). This request goes through the store buffer, in order to ensure that all stores from the threadlet have drained to the SSB first. Then, the request is sent to the SSB, which initiates the commit as soon as possible (see Section 3.10.6), and responds to the processor with a "commit initiated" message. Once this has happened, the reattach instruction is ready for its final commit in the pipeline. It is processed a second time in the commit stage, taking up a commit slot, and initiating the shutdown of the threadlet and the recycling of the context. This process takes a few cycles, to ensure that all pipeline stages are left in a clean state. This includes draining out any stale (squashed) instructions, and ensuring that no entries remain in any of the structures and all counters reflect this too. Once this has finished, the context is marked as *idle*, ready to accept a new epoch.

Sync instructions

A matching sync instruction causes an exit from the parallel region. Once it gets to the front of the ROB, it immediately squashes all successor epochs, without restarting them. This happens regardless of whether the threadlet is speculative or architectural. If it is speculative, then it could still be squashed due to an inter-threadlet true memory conflict, and then the region exit would not occur. However, if this happens, we would squash and recycle all successors upon the squash triggered by the conflict, so we may as well squash and recycle them now to free up resources.

Following this, the sync waits until its threadlet becomes architectural. When this happens, the sync is ready to commit. When it does, it clears the active region ID, and resumes execution in sequential mode. Statistics that were reset on region entry may be read here to yield statistics data for a single region invocation.

Register state merging

As discussed in Section 3.2.4, in order to help the compiler's (and the microarchitecture's) job, the architecture allows register data flow from anywhere inside a region to instructions after the sync instruction. To implement this, the microarchitecture needs to merge the register end state of each epoch. Specifically, when committing a reattach or sync instruction, the corresponding threadlet is architectural, and therefore its predecessor has finished. When the predecessor retired, it sent over its final architectural register state to the current threadlet. Now that the current threadlet is retiring – or preparing to run instructions outside of the current region (after sync) – the microarchitecture works out the new (merged) architectural register state at the end of the current threadlet.

The inputs to this logic are the current threadlet's register write set, the predecessor's architectural end state and the current threadlet's end state. Any registers written in the current epoch are taken from the current threadlet's end state, and the other registers are inherited from the predecessor's architectural end state. The currrent threadlet's register rename map is updated to reflect the merged end state. After during this step, any now-unused physical registers are freed. In the case of register merging on sync, the frontend rename map of the current threadlet is also updated, before the threadlet can resume execution outside the region.

3.10 Memory system

Speculative state is never observable in the coherent memory system, as exposing it would have a significant impact on the memory model (i.e. consistency and coherence model) of the architecture. Instead, speculative state is buffered in the speculative state buffer (SSB) until the corresponding threadlet is committed to architectural state. There is a well-defined commit point in time, when all buffered state becomes architecturally visible in an instant. This is achieved by incrementing a single *architectural epoch counter* in the SSB, which causes the SSB to respond to snoop requests from other caches, thus reporting up-to-date values. Therefore, the SSB may hold both architectural and speculative values. While

architectural state stored in the SSB forms part of the coherent cache system, speculative state is hidden from the rest of the system.

3.10.1 Assumptions about the architecture

The requirements for the SSB depend on the restrictions imposed by the memory consistency and coherence models. Here, I assume the memory model used in the Armv8 architecture, which is similar to many reduced instruction set computer (RISC) architectures. The design is for normal (i.e. not device or otherwise restricted) memory. If a write to other type of memory is encountered in a parallel region, then we can sequentialise execution (by squashing speculative threadlets) to avoid issues.

Only a single up-to-date value may exist for every cache line at any given time across the coherent part of the memory system, which comprises the caches and main memory (but not the store buffers, merge buffers, load-store queues, etc.). Furthermore, the reordering of memory operations is restricted by the consistency model. The Armv8 consistency model [65] generally permits speculative reads of memory, as well as the eliminination and merging of some operations. Reordering is permitted between any operations (reads or writes), however a handful of restrictions are placed on reordering. Firstly, sequential correctness must be maintained. Secondly, operations to overlapping locations must be observed in order, irrespective of operation type. And thirdly, operations from address-dependent instructions must be observed in order.

The processor may physically still process operations out of order in any of these scenarios, but doing so creates a hazard, which must be resolved as if processing happened in order, and it cannot be observable to any other core in the system. For example, if A and B are two load instructions targeting the same address X, where A comes first in program order, then the processor could still perform B first (for example because it has not calculated the address for A yet), but if another core stores to address X before A reads the value, then a violation has occurred, which may be resolved by squashing and re-executing the load instruction B.[11]

I assume a MOESI coherence model here for simplicity of explanation, but other models can likely be supported similarly.

3.10.2 Speculative state buffer overview

The SSB is a buffer that handles multiple important functions to enable microarchitectural in-core thread-level speculation. Most importantly, it buffers speculatively stored memory values, and serves them to speculative load instructions. The SSB also organises epoch commit to architectural state, and buffers now-architectural values until they can be drained to the memory system. Speculative values in the SSB are not part of the coherent memory system, but architecturally committed values (which have not yet been written back) are. Thus, the SSB snoops coherence traffic, but only responds to requests that overlap with architectural values, providing these to the requestor. This dual mode of operation allows for

[11] Note that this is an ordering constraint, not a data flow one. For overlapping loads A, B and store S, all orderings (SAB, SBA, ASB, ABS, BAS) except BSA are allowed. The architecture forbids the *observable* reordering of the two reads.

constraining speculative state to speculative threadlets within the same core, while allowing for atomic commit, respecting the consistency model, as described in Section 3.10.6.

Further to this, the SSB also maintains correctness by resolving hazards. The SSB's design eliminates write-after-write and write-after-read hazards by keeping speculatively written state private to the epoch that created it. By providing the illusion of atomic commit to the rest of the system, the SSB can facilitate conformance to the architecture's consistency model (including the elimination of read-after-read hazards if required). Read-after-write violations between speculative epochs are caught by the conflict detector of the SSB, and resolved by squashing and restarting the younger epoch.

The SSB described here builds on prior work in hardware transactional memory and versioned caches for thread-level speculation, as discussed in Section 2.5, but extends both. Similar to prior versioned cache designs [25, 74, 76], the SSB supports holding multiple speculative versions, and handling the ordering and dependences. However, note that the SSB keeps speculative state within a single core, and – like hardware transactional memory systems – still complies with typical coherence and consistency requirements of modern multicore systems. Doing so enables interoperability with traditional multithreading across cores. This compatibility is very important from a deployment perspective.

Additionally, the SSB supports sub-cacheline granularity conflict checking and operations, thus eliminating false conflicts, and enabling support for handling write-after-write dependences without resulting in a squash. For this, high granularity is crucial, as write-after-write dependences can only be safely ignored if the later write in program order fully overwrites its granule, or else the two (or more) inconsistent versions of the same granule would need to be merged. For example if write W_1 writes the start of a cache line, and W_2 writes the end, then the correct final version of the line (after both writes) includes the output of W_1 at the start, some unmodified bytes, then W_2. In comparison, in the SSB, W_1 and W_2 typically end up touching separate granules (thus not creating a hazard) or they entirely overwrite a granule, in which case the final result is the result of W_2.

3.10.3 Buffering speculative state

The SSB holds speculatively written values in a *granule cache*. In the version presented here, the granule cache consists of multiple independent slices, one for each epoch. Each slice contains a set of small (e.g. byte-level) *granules*, which together form the write set of the corresponding epoch. To keep the overhead of metadata (and write latency) reasonable, small granules can be implemented using larger (32 or 64 byte) cache lines, and fine-grained, granule-level masking.

Reducing the granule size reduces the number of conflicts resulting from false sharing. This is because any partial writes to a new granule require the rest of the bytes to be fetched from the memory system. Since later writes to those bytes of memory by previous speculative epochs could thus result in a (partially) stale value in this younger granule, such partial writes need to be treated as a full-granule read followed by a write. This additional read may trigger a conflict in the conflict detector, even if the two epochs never accessed the same bytes.

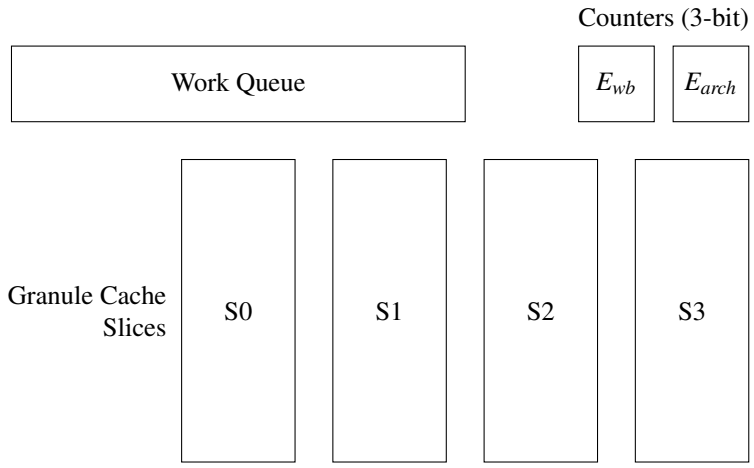


Figure 3.10: The SSB consists of a work queue, two 3-bit counters to track the status of epoch commit and writeback, and four granule cache slices (one for each speculative epoch).

3.10.4 SSB microarchitecture

The SSB holds speculatively written values in a *granule cache*. The optimal low-level design and cache organisation for the SSB is beyond the scope of this thesis, but I present one possible implementation here. The top-level structure of the SSB is shown in Figure 3.10.

Logically, the granule cache consists of multiple slices, each of which stores a set of small (e.g. byte-level) *granules*, which together form the write-set of the corresponding epoch E. Additionally, architectural values from past epochs that have not yet finished writeback are also stored in the granule cache. Specifically, the granule cache slice for the the speculative epoch E may also hold architectural values from epoch $E-4$. Thus, this SSB is capable of holding data from 4 speculative and 4 architectural epochs, thus 8 in total.

The SSB uses 3-bit epoch IDs to track epochs. The E_{wb} counter stores the ID of the last epoch to finish writing back fully, and E_{arch} refers to the epoch to last commit to architectural state. A specific epoch E will always be stored in slice $S_{E\%4}$, and thus these two counters serve for tracking the order between granules coming out of the cache, as well as free epoch numbers. Specifically, using the 3-bit (overflowing) addition operation $\oplus$, we can say $E_{wb} \oplus 1, E_{wb} \oplus 2, \ldots, E_{arch}$ refer to architectural epochs (thus granules tagged with these IDs are architectural), and $E_{arch} \oplus 1, \ldots, E_{arch} \oplus 4$ refer to speculative epochs. Moreover, the ordering between epochs is $E_{wb} \oplus 1 \prec E_{wb} \oplus 2 \prec \cdots \prec E_{arch} \prec \cdots \prec E_{arch} \oplus 4$. Finally, the number of architectural epochs not yet written back is $E_{arch} \ominus E_{wb}$, which can be used to delay epoch commit to limit their number.

Figure 3.11 shows one possible implementation for one of the four granule cache slices. The slice consists of a set-associative main cache, storing the granules, two counters and an optional victim cache.

The lines in the main cache are 64 bytes long. The bottom 5 bits of the address are used as an offset into the line, the next 5 bits ($A[10:6]$) are used to index the associative set, and the top bits ($A[51:11]$) of the address are used for the tag along with the top bit of the epoch ID, $E[2]$. The epoch ID is part of the address of the line, so the cache might have a line for the same address from each epoch. Note that the bottom two bits of the epoch ID are implicit,

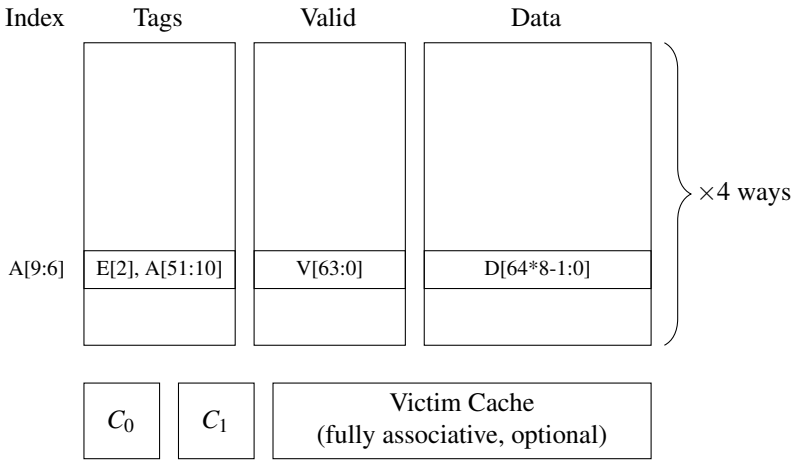


Figure 3.11: Each granule cache slice consists of an associative cache, an optional victim cache to increase associativity, and two counters to track architectural and speculative occupancy.

as they are the same as the slice number. Note that for each slice, $E[2] = 0$ and $E[2] = 1$ will correspond to an architectural and a speculative granule, with the mapping determined by the value of E_{arch}. The slice also contains two counters, tracking the number of lines from each of these epochs. These are used to track the completion of writebacks to memory. When writing back epoch $W \triangleq E_{wb} \oplus 1$, then the writeback is done (and E_{wb} can be incremented when $C_{W[2]}$ in the slice storing epoch W hits 0. In addition to 64 bytes of data, each line also contains a valid bit-mask V. With one-byte granules, the mask has 64 bits. An all-zero valid mask means the line is not present.

When allocating a new line, the cache first tries to claim an empty line (with its valid mask containing all zeros). If this is not possible, an architectural line can be pushed out by writing it back early. If this fails, there is an issue: since speculative lines cannot write back to the memory system, they cannot be evicted without squashing their epoch. To handle such conflict misses, a small fully-associative cache (or *victim cache*) can be added to the granule cache to boost its effective associativity. If no free lines are left in a set, the new line is instead allocated in the victim cache. The victim cache is looked up in parallel with the main cache. If the victim cache is also full, the epoch has to be squashed, or suspended until it becomes architectural. The prototype does not model associativity, as this is a low-level design detail.

The operation of the SSB in response to different requests is described in the next section.

3.10.5 Implementation of operations

The SSB observes all operations sent from the core to the first-level data cache, each tagged with the corresponding epoch ID. Certain operations are intercepted, delayed and modified to provide correct results. The details depend on whether the operation is from a speculative or architectural epoch, and whether it is a load or a store.

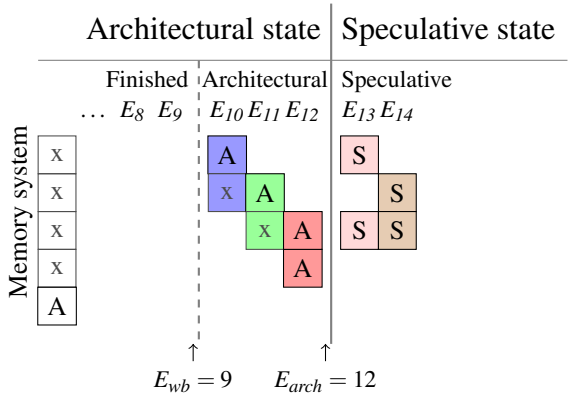


Figure 3.12: Classification of epochs and their granules. Granules marked with x are stale and not observable, those marked with A are architectural (i.e. externally observable and coherent), and those marked with S are speculative and only observable to threadlets in the same core running younger epochs. Epochs 9 and earlier are finished and do not have any granules in the SSB, epochs 12 and earlier are architectural, while $E13$ and $E14$ are speculative.

Classification of operations The main idea is that we need to intercept both the request and the reply for speculative operations, but we would like most architectural operations to proceed uninhibited, bypassing the SSB altogether. However, there are a few edge cases that need to be handled for correctness.

Firstly, we cannot rigidly classify operations into the architectural and speculative categories, as their classifications may change while they are in-flight. Secondly, the possible existence of architectural state in the SSB also poses challenges.

The SSB uses its E_{wb} and E_{arch} counters to determine whether an operation from a given epoch E_{op} can bypass the SSB, or whether it needs to be intercepted.Since $E_{wb} \leq E_{arch}$, and operations can be separated into three categories, based on the relationship of their epoch number, E_{op}, with respect to these pointers. If $E_{op} \leq E_{wb}$, then the operation is from a *finished* epoch, and there are no granules in the SSB that have been written by preceding operations in program order. This may be the case after a partial commit, or in sequential mode. If $E_{wb} < E_{op} \leq E_{arch}$, then the operation is from an *unfinished architectural* epoch. Thus, it is an architectural operation, but it may need to observe SSB granules. Otherwise, $E_{arch} < E_{op}$, and therefore the operation is from a speculative epoch. Each type of operation is handled differently, as described below.

Operations from finished epochs These operations do not need to interact with the SSB, and they can proceed without being delayed whatsoever. This is the common case for architectural operations, and it is the basic mode of operation in sequential mode. Unless there is no active region at all, store operations in this category still update the conflict detector (asynchronously), because they could be the source of a true (read-after-write) conflict, between their epoch and a younger, speculative epoch.

Stores from unfinished architectural epochs The store request proceeds to the level-one cache, but if the SSB also has overlapping architectural data, then either the overlapping data bytes need to be discarded from the SSB (due to the newer architectural write) or the write needs to be merged with the line in the SSB, and written back together.

To look up architectural granules for a given address, the SSB simultaneously looks up the granules in all slices that contain architectural data (i.e. the slices for $E_{wb} \oplus 1, \ldots, E_{arch}$) with the appropriate epoch IDs, and takes the newest version of the line.

The architectural write could also be merged together with all granules waiting to write back, and written back as a whole cache line, but this is not implemented in the current version.

Loads from unfinished architectural epochs These loads can also proceed uninhibited to the memory system. However, parallel to the cache read, the SSB checks if it has any overlapping architectural granules for which the writeback has not yet been started. If so, it will modify the response to the load request (once the cache sends it back) before serving it to the memory unit in the CPU pipeline. The version presented in this chapter performs an architectural lookup in the SSB (targetting all relevant slices), and forces write-back to the cache before re-reading the load. The SSB logic required here is the same as the logic when a read request is snooped from another core.

Speculative stores Speculative stores do not write values to the coherent memory system. Instead, they are intercepted by the SSB, and inserted into the operation queue, where they stay until they can be processed. Speculative stores wait for any overlapping blocked (waiting) operations from the same epoch.

When able, the SSB sends a coherence message requesting the affected cache line in the *exclusive* state, as discussed in Section 3.10.6, in order to facilitate atomic commit of the epoch to the memory system later. When the response to the prefetch is received, the written value is stored in the SSB, and a response is generated. Any granules that already exist for overlapping addresses for the same epoch are overwritten, and a new granule is created for all other addresses.

Speculative loads Speculative loads are intercepted by the SSB and queued. The load is only dispatched once all overlapping, in-flight (or queued) writes from the same epoch have been processed. This ensures correct ordering with respect to stores from the same threadlet.[12]

Once they clear the operation queue, speculative loads are sent to the memory system as normal. The consistency model allows speculative loads to be observed system-wide, as these are already commonly used when exploiting ILP. Sending the load request to the memory system achieves two things. Firstly, it obtains the current architecturally visible value of the memory location, and secondly, it brings the cache line into the level-one data

[12] If there are prior stores in the load-store queue or the store buffer, the CPU pipeline would ensure correct ordering. Otherwise, the load will wait for all remaining prior stores in the SSB. Future stores are only released to the SSB after they have committed in the pipeline, which is after the load commits.

cache in the shared state, thus the SSB can monitor any future writes to the cache line from other caches, as they will not have the line in the exclusive or modified state.

While the load is processed in the level-one cache, the SSB also performs a lookup for that address from the given epoch, and it returns both the data and the valid bitmask for the line. Together with the (architectural) data read from the cache system, the SSB can construct obtain the most up-to-date data – as seen from the load's epoch – using simple combinational logic. Since any data read from the SSB is more recent than data read from the memory system, the SSB simply takes the granules identified by the bitmask from the data read from the SSB, and it takes the remaining bytes from the value received from the memory system. This logic is simple per-granule multiplexing, and it is parallel between the different granules of the request. Therefore, it does not add a significant amount of delay to the load response, and may be possible to perform in the same cycle.

3.10.6 Coherence and epoch commit

Note that the above methods implement correct ordering (and thus atomicity) for the microarchitectural threadlets derived from the same (architectural, OS-visible) program thread/process, which all run inside the same core. However, to support traditional multi-threading across multiple cores (in addition to transparent threadlet-based parallelism), we need to take additional care. In particular, other cores must not observe any illegal reordering of memory operations as defined by the architecture's memory model. To achieve this, operations need to appear in order to the coherence protocol.

A naïve implementation of this would be to buffer and replay all speculatively executed operations (stores and loads) in order, and ensure that loads observed the same values. If they did not, then the threadlet would need to be restarted from the first failing load. However, this requires a large amount of state tracking, as well as frequent checkpoints (on every load).

Instead, we will relax this slightly by exposing multiple operations simultaneously. Due to atomicity with respect to operations from other processes, we do not need to maintain order between operations exposed in this way, and we can compress multiple reads/writes to the same location into a single read/write.

Let us first understand the mechanism on a high level, and then discuss the specifics in detail.

Overview

Following past approaches implementing multi-line atomic commit for hardware transactional memory [34, 38], we use the coherence protocol to obtain all read cache lines in a *readable* state, and all written cache lines in a *writeable* state. The idea (as detailed below) is that we keep all the lines in these states throughout the commit process, and this yields in two important properties. Firstly, all operations can be performed without further coherence state transitions (thus uninhibited, and without a chance for deadlock), and secondly, the whole set of operations appear atomic to any other observers.

The first part is easy to see, as we have the correct permissions for all the reads and writes, since lines read are in the readable, and lines written are in the writeable state. Note that lines that are writeable are also automatically readable by design.

The second part relies on the single-writer, multiple-reader property of the coherence protocol (such as MOESI). That is, lines in the readable (e.g. *S: shared*) state in one core are not writeable by any other core, and lines in the writeable (e.g. *M: modified*) state are not readable or writeable by others. Since committing writes target writeable lines, they can only be observed once a state transition occurs (e.g. to the *shared* state), which we only allow after commit finishes. Similarly, reads target readable lines, so other cores can only write to these lines after a state transition, after epoch commit finishes.

Therefore, atomic commit is guaranteed if we can ensure that the states are held throughout the commit process. However, delaying state transitions (snoop requests) significantly could have an extremely detrimental effect on system performance. Furthermore, we need to ensure no deadlocks can occur due to the way we acquire lines. To do this, we build the read and write sets gradually, over the lifetime of the epoch, we squash if permissions are ever lost, and we use the ability of the SSB to hold architectural lines to implement instantaneous commit (once all the targeted lines are acquired) once all the permissions are held, and then flush them to the memory system over time. The details are discussed below.

Implementation

While an epoch is speculatively running, the SSB intercepts memory operations. Reads are logged in the read set in the SSB, and forwarded to the memory system. The value returned by the L1 cache is used as the base for the value returned from the SSB. With the appropriate flags, this also brings the read cache line into the L1 cache, into a readable state. The SSB can then monitor state transitions on this line by snooping coherence traffic. If the cache line ever transitions away from the readable state, the SSB notes that readable permission has been lost. If permission is lost, another core could write to the line, thus changing the correct input value to the epoch, and so the epoch can no longer commit. The SSB will notice this by snooping the bus, prevent epoch commit, and initiate squashing to restart speculative execution from the beginning.

Note that the line does not need to remain in the L1 cache itself (it can be evicted due to space or a line conflict). So long as no other core performs a read-exclusive access (or non-cacheable write), we can be sure that the value will not remain unchanged.

The mechanism is similar for speculative writes. The difference is that – since writes are buffered in the SSB – we do not need the contents of the cache line immediately. Thus, speculative writes can simply issue a prefetch into the exclusive (writeable) state in the background. The line only needs to arrive by the time the epoch is committed. Such requests are sent off in the background, and epoch commit is delayed until all lines are successfully acquired. Note that the microarchitecture is allowed to squash the epoch at any point until commit, thus any failures in acquiring lines can be safely resolved (by squashing speculative work).

Once all the lines are in the required states (i.e. all reads have finished, all write prefetches have succeeded, and no permissions got lost), commit can happen. To ensure that commit is atomic, the SSB makes all updates externally visible simultaneously (as this ensures permissions are not lost during the commit process). To do this, it increments the architectural epoch counter, and monitors coherence messages for lines it has buffered architectural writes

for. If another core attempts to read such a line, the SSB will immediately drain the buffered write(s) for that line, and it briefly delays the snooped request until this happens.

Freedom from deadlocks

Although other designs are possible, I opt for a design that never tries to re-acquire lines. If necessary permissions are ever lost, the corresponding speculative epoch is squashed (and restarted). This policy ensures that no deadlocks can occur, because the architectural threadlet can always continue to make progress.

Firstly, in a multi-core environment, speculative threadlets in one core cannot stop the architectural threadlet in another core from acquiring lines it needs. Secondly, within the same core, the architectural threadlet can never be forced to wait for a speculative threadlet indefinitely. The only case where a (temporary) wait can occur is if the architectural threadlet has reached its final reattach, and it is waiting for its successor to commit. If commit succeeds, progress continues. Otherwise, speculative work is discarded and the successor restarts from the beginning. Since the squash discards all speculative work, the successor will have no buffered speculative reads or writes. Thus, it can trivially commit immediately (since it has nothing to commit). The commit recycles the older threadlet and turns the successor architectural, at which point its progress is guaranteed as before.

Implementation on HTM-capable systems

If the system already supports hardware transactional memory, then the overhead of implementing the SSB can be lowered. There are several options for doing this.

Firstly, the SSB and the transactional cache [34] could be merged. The SSB design shown here can buffer speculative updates, gather the read and write set of the transaction, obtain the right cache lines in the correct states, handle aborts, and facilitate writeback on commit in the same way it would for speculative epochs. Thus, the already-endured cost of the transactional cache reduces the *additional* cost of the SSB functionality. Note that transactions within speculative epochs can be supported, and these still make sense. Specifically, speculation affects sequential performance, and the purpose of transactions is to enable lock-free traditional multithreading. If a speculative epoch touches shared data, those accesses still need to either be protected by a lock or wrapped in a transaction. The design proposed here could artificially delay the commit of an epoch which has an active transaction until the transaction commits, whereas a transactional abort will automatically squash the epoch.

The second option is to implement a more light-weight SSB, which supports multi-versioning and conflict checking, but not coherence-safe commit by itself. This SSB could then submit its sequence of speculative operations to commit to the HTM implementation to commit to the memory system as the previous threadlet reaches its final reattach. Note that the order of operations does not matter due to the atomicity of transactions, and the committing threadlet can continue running in the meantime (attaching any ongoing memory operations to the transaction). If transactional memory yields an abort, the threadlet is restarted.

Note that in both of these cases, intra-core conflict granularity is fine-grained and resolved by the SSB, while inter-core conflicts are evaluated at cacheline granularity, utilising the coherence system (either by the SSB or the transactional memory implementation).

3.10.7 Squashing

When epochs get squashed, the SSB needs to clear out all stale granules, and ensure that they never get written back to the memory system. If an epoch is restarted, the SSB needs to make sure that state created after the restart is separated (and not wiped as part of the squash). Care must be taken here to guarantee correct operation even in timing edge cases, including stores about to write back from the store buffer and a commit request arriving shortly after the squash.

To achieve this goal, the SSB stores a *retry number* for each epoch, which serves as a tag, and it is incremented on a squash of the epoch. Each memory request carries a retry number, which is checked against the retry number in the SSB. Stale operations are discarded. Doing so allows such operations to clear from the system without changing state. The retry number can be implemented using a single bit, marking each retry of an epoch with one of two colours in an alternating fashion.[13]

Furthermore, a per-epoch *invalidate* flag is set, marking all granules from the given epoch as stale. This prevents the squashed epochs from writing back to the cache under any circumstances. In the subsequent clock cycles, these granules will be invalidated in the background, using spare bandwidth. Once all of them have been invalidated, the stale flag is reset.

While the invalidate flag is set, stores from after the restart need to be delayed, as allowing them to write to the SSB would mix old and new state, and possibly cause the new state to be discarded. However, note that in practice quite a few clock cycles will elapse between receiving the squash signal and the first post-restart write. This is because the CPU pipeline needs to squash and clear out all in-flight instruction and reset its state, then redirect and restart fetch, process and (speculatively) commit a store instruction, then insert it into the store buffer, and write it back from the store buffer to the SSB.

The simulated model does not handle this case, and instead approximates squashing as an instantaneous clearing out of all granules. The details are given here for completeness. If required, retry numbers could also be maintained for each granule, although the cost of keeping this extra metadata would be unlikely to pay off.

3.11 Conflict detection

True (read-after-write) memory conflicts between epochs must be detected, in order to issue corrective action (squashing the offending epoch). The memory sub-system (see Section 3.10) handles other types of dependences: sequential atomic commit and multi-versioning in the SSB resolves read-after-read, write-after-read and write-after-write hazards.

[13] Assuming in-flight requests to the rest of the memory system are tracked separately, and responses to stale requests are discarded.

Algorithm 3 Conflict detection logic

Require: Read and write sets $\text{RD}(E)$, $\text{WR}(E)$ for each epoch E

```
1:  function SPECULATIVEREAD(Epoch E, Granules G)
2:      Fwd ← G \ WR(E)    // Forwarded: read before any writes from the same epoch.
3:      RD(E) ← RD(E) ∪ Fwd    // Granule read by epoch iff forwarded.

4:  function WRITE(Epoch E, Granules G)
5:      WR(E) ← WR(E) ∪ G

6:  // Epoch E − 1 is detaching a new epoch E.
7:  function DETACHNEWEPOCH(Epoch E)
8:      // Remember the current architectural epoch.
9:      // This is the oldest epoch that ever ran in parallel with E, and thus can conflict with it.
10:     OLDESTPARALLELEPOCH(E) ← ARCHITECTURALEPOCH

11: // Epoch E − 1 has reattached. Epoch E is about to become the oldest epoch.
12: function COMMITEPOCH(Epoch E)
13:     // Check if E's speculation is correct.
14:     for all e from OLDESTPARLLELEPOCH(E) to E − 1
15:         if WR(e) ∩ RD(E) ≠ ∅    // E observed stale value!
16:             SQUASH(E)    // Squash and restart E, recycle E + 1, E + 2, ...
17:             // This discards all speculative state of E.
18:             break
19:     // Either there was no conflict, or we have just discarded all speculative state and restarted E.
20:     // Thus, we can commit and expose E's state architecturally.
21:     ARCHITECTURALEPOCH ← E
22:     // We can free up WR(e) for all e < OLDESTPARALLELEPOCH(E+1) (or all e if E + 1 doesn't exist).
```

3.11.1 Checking logic

Logically, conflict checking uses the read and write sets of each epoch to identify conflicts.

When an operation is serviced by the SSB, conflict checking is performed as shown in algorithm 3.

For architectural reads, nothing needs to be done, as the value read is guaranteed to be correct by the memory system. For all write operations, we update the write set of the corresponding epoch. The write set of an epoch E is retained even after E has committed, and even after it has reattached, as it may need to be checked against the read sets of epochs $E+1$, $E+2$, etc. The write set can be recycled once all epochs that ever co-existed with E have been committed. This is tracked by the $\text{OLDESTPARALLELEPOCH}(e)$ values. The latest that $\text{WR}(E)$ may be needed is when epoch $E+N-1$ commits, where N is the number of threadlet contexts in the system (e.g. 4).

For speculative reads, we update the read set of the epoch. Not all granules read by the read operation itself are added to the epoch's read set. Specifically, any granules that have been overwritten in the same epoch are excluded, as they cannot lead to a read-after-write hazard between epochs, as the younger write *hides* the older one from the read.

An operation that only partially overwrites a granule (e.g. only writes to the first 4 bytes of an 8-byte granule) is handled as a read followed by a write to ensure correctness. False aliasing from this can be improved by reducing the granule size.

When an epoch is (partially or fully) committed – that is, when its predecessor successfully reattaches – we perform conflict checking. We are looking for reads in the committing epoch that may have observed stale values. To do this, we compare its read set against the write sets of all previous epochs that co-existed with the current one at any point in time. Writes from epochs that reattached before the current one was launched do not need to be checked, as they are before the current epoch in program order, and they were correctly performed before the present epoch's operations in the memory system.

The commit-time conflict check has two possible outcomes. It either passes or fails. If it fails, we squash the threadlet executing the epoch, discarding all speculative state, including memory operations, register values, the program counter, and any successors that may have launched. The state of the threadlet is reset to the start of the epoch by reloading the starting checkpoint, and it restarts execution from there. If conflict checking passes, current speculative state is retained. In either case, all previous epochs have reattached, and so the threadlet is now the oldest in the system (as its predecessor has just reattached), and it is free from any incorrect speculative state. Therefore, it can safely become architectural. Bumping up the ARCHITECTURALEPOCH counter makes the SSB expose any granules produced by the current threadlet to the coherent memory system, thus they immediately (and atomically) become architecturally visible.

As a secondary mechanism, the conflict checker also snoops coherence traffic, and monitors if appropriate ownership of granules in the read and write set is lost due to operations from other cores (or hardware threads), as described in Section 3.10.6.

3.11.2 Read and write set implementation

There are multiple ways to implement the set comparison operations required here, and the exact implementation is beyond the scope of this thesis.

As a starting point, we could use Bloom filters [6], as used in prior work [92]. Bloom filters provide a compact encoding of sets, and enable efficient, but approximate membership and intersection tests. The approximation is safe here, as it is one-way, with false positive outcomes allowed but no false negatives. For example, the output of an intersection test is either *'may intersect'* or *'does not intersect'*. For 8 KiB read sets, with a granule size of 8 B, we have $1,024$ set elements. Thus, a 4 KiB ($32,768$-bit), 22-way Bloom filter can provide an approximate[14] false positive rate of 2.1×10^{-7} per membership test. This works out to less than 0.2% false positives per epoch (assuming $10,000$ membership tests as an upper bound). This means only about one in 500 epochs need to be squashed due to the approximate nature of Bloom filters.

Although these filters add significant area as is, the implementation described here is only a crude first approximation. We can expect an optimised implementation to exhibit much smaller area. For example, the write sets are already stored in the tag array of the

[14]Using the well-known approximation $P_{error} \approx (1 - e^{-kn/m})^k$ from prior work [52] (with $n = 1,024$ set elements, $m = 32,768$ filter bits and $k = 22$ ways).

SSB's granule cache. If we also store read sets similarly, then we can perform exact checking on-the-fly in the background (without blocking threadlets), perhaps using less-accurate bloom filters to reduce accesses to these structures. Alternatively, we can track read and write sets (e.g. using Bloom filters or content-addressable memory) at cache-line granularity, and trigger more fine-grained checking in case the cache-line-level checking yields a conflict (to see if accesses actually overlap). There is ample space for optimisation, but I do not explore this here, as the optimisations discussed in Chapter 4 change the requirements, and necessitate a reimplementation of the conflict checker anyway.

In the headline figures, I assume no additional delay for conflict checking, assuming that most of the work can happen in parallel, and I also investigate the effects of increasing conflict checking latency in the evaluation, Section 3.12. I also do not model the inaccuracies of bloom filters, as expected number of resulting squashes is insignificant compared to squashes due to true conflicts.

3.11.3 Register spill and fill support

As mentioned in Section 3.2.4, while general register data flow between the body of one epoch and any future epochs leads to a conflict and triggers a squash, the architecture permits limited data flow for register spill and fill patterns without squashing. In particular, squashing can be avoided if an epoch updates the value of a register, and the successor has only read that register's starting value to spill it to memory. The idea is to handle the conflict without squashing, by updating the value spilled to memory (if not overwritten by something else), and relying on the register state merging logic to update the final value of the register (after it has been filled back).

Listing 3.2 shows a typical spill and fill pattern. Suppose the register *r7* is a callee-saved register in the calling convention, the function FOO will have to save (spill) its value at function entry, and restore (fill) it at function exit if it may clobber (overwrite) the value inside. The starting value of *r7* will not be accessed, apart from the pair of store and load instructions shown here.

Note that the pattern observed (at run time or compile time) may be more complicated in reality. FOO may be recursive, it may be called multiple times, or it may call other functions that also spill the same register. Furthermore, multiple calls may be made in sequence too

Listing 3.2: Typical register spill and fill example. Register *r7* is spilled to the stack.

```
1   body:
2     ...
3     call foo()
4     ...
5
6   foo:  // Function
7         str r7, [rsp + 0x64]      // Spill
8         ...
9         ldr r7, [rsp + 0x64]      // Fill/restore
10        ret
```

(e.g. if FOO is called twice in the body). These multiple spills may or may not save and restore the same value.

Compile-time limitations One option for handling these would be at compile time. This can be done by finding a safe approximation for the set of registers that may be modified inside the body. Then, these registers can be saved (to the stack) at the start of the body, then restored at the end, just before the reattach point. Doing so ensures that the register state between the start and end of the body is exactly identical, thus spill-fill patterns (and other reads) in the successor never observe stale values (even on dead registers).

However, this adds extra code and results in sequential performance degradation, especially because of the need for a safe over-approximation of the set of modified registers. In particular, if function calls are present in the body, this would include all caller-saved registers, since the callee may clobber these, leading to changes in the body (which might conflict with a spill-fill in the successor). This can only be avoided if expensive inter-procedural analysis can be performed, and all potential callees are available in the same compilation unit (or during link-time optimisation).

Spill and fill patterns can be tracked at run time by the microarchitecture, as described below. Then, if a true (non-spill) read-after-write register conflict is detected, the reader epoch can be squashed, otherwise all (still live) spill locations that contain stale values (due to spill read-after-write conflicts) can be silently fixed up just before epoch commit in the SSB, in order to faithfully preserve sequential semantics.

Spill and fill pattern at run time The microarchitecture monitors the instruction stream to find instructions that are consistent with the spill-fill metaphor, and discounts them from the register read and write sets of the epoch. The set of patterns that are consistent (and thus recognised) are defined in this section.

Figure 3.13 shows a finite state machine that defines the status of a register R with respect to a single spill location M. The top four states in the diagram keep track of where the original value of the register is currently stored, whether it is in the original register R, the memory spill location M, both or neither. The value starts in only the register, hence the enry point is the $\{R\}$ state.

The bottom '*observed*' state means that the register's starting value affected the epoch in a more complex way, thus it has been added to the epoch's read set. Specifically, this happens if the original value was copied to a different register, or if it was used as an input to an ALU instruction or to calculate a memory address. These actions are called 'non-spill reads' in this section. This can be regarded as the failure state for spill pattern recognition. Once a register is added here, it can no longer transition to another state, and conflicts on registers in this state are not handled specially.

From here, we can derive the state machine transitions. Any non-spill read from a copy of the original value (whether it is in memory or register) takes us to the *observed* state. A store instruction copies the value from the register to memory, while a load copies it from memory back to the register. Neither of these two have any effects if the output (register or memory location) already contains a copy. On the other hand, any instruction overwriting the register or memory eliminates the corresponding copy. If all copies are eliminated,

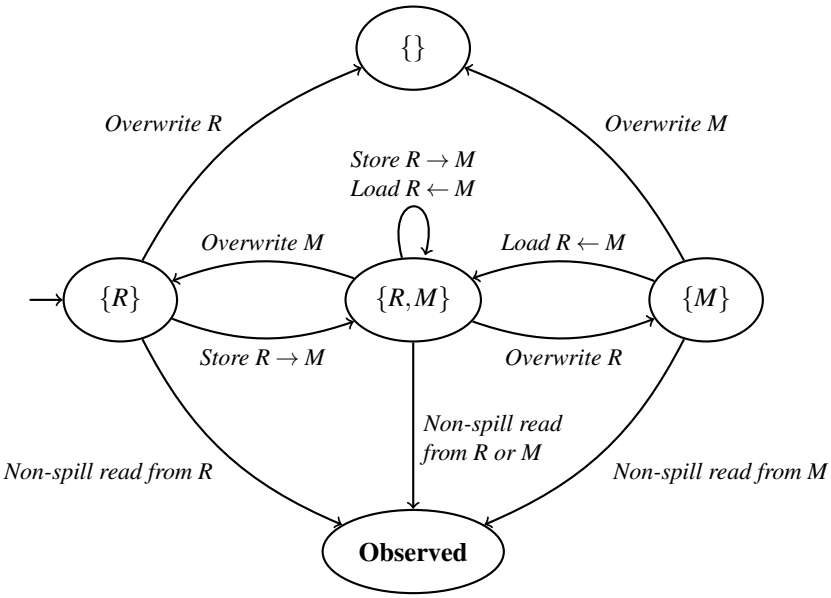


Figure 3.13: Logical status of a (suspected) spilled register R (for a single spill location M)

the register starting value is *dead*, and it cannot be recovered, thus we stay in the $\{\}$ state permanently.

Multiple spill locations While the above method only tracks a single memory spill location M, in reality the same starting value may be spilled to multiple locations. We can permit and track such behaviour by maintaining a set of memory locations and a single boolean flag per register (in addition to tracking the register read set). The set contains all the spill locations in memory that currently contain a copy of the register's starting value, while the boolean flag determines whether the register itself contains its own starting value.

When observing the stream of instructions, we can easily keep these structures up to date. If we write to the register, we clear the flag. If any memory store overwrites a spill location, that location is removed from the spill set. If a memory store reads a register whose flag is set, then the target address is added to the spill set. If a memory load reads a location in the spill set and writes it to the corresponding register, then the flag is set. And finally, if a memory location in the spill set is loaded to any other register, or if a register is observed in any other way while its flag is set, the register is added to the read set of the epoch, and all of its spill state is cleared.

This is equivalent to running multiple copies of the above state machine simultaneously, one per possible spill location, with the *Load* $R \leftarrow M$ transition expanded to *Load* $R \leftarrow M'$ where M' is a memory location such that the state machine (R, M') is in the state $\{M'\}$ or $\{R, M'\}$, adding the same transition from $\{\}$ to $\{R, M\}$, and considering the value as *observed* if any of the state machines are in the *observed* state. A combined state machine is trivial to derive, but the number of states and transitions is large.

Hardware implementation In the microarchitecture, we can add a bitmap of architectural registers[15], and a small content-addressable memory (CAM) structure, storing context/epoch ID, register number and memory address 3-tuples. One register may have multiple mappings, but each address may only appear once. The structure should be addressable by register number or memory address, and looked up on all load and store instructions from the epoch as described above. Entries containing a register that is in the read set can be automatically considered invalid.

A further potential optimisation – to reduce the number of CAM lookups – is to add one bit of metadata to each TLB entry. If this bit is unset, we know that the given page of memory contains no spill locations. This would likely be unset for non-stack pages (which is most of the pages). Alternatively, we can limit checking to the region of memory that is likely the stack, for example by only monitoring the range around the base pointer and/or the stack pointer (e.g. $\texttt{rsp} \pm 512$ bytes).

Fixing up values in memory and registers When an epoch reaches reattach, its final register state is available. When the reattach request reaches the SSB, this is merged with the successor epoch. If the value of any registers changed compared to the starting checkpoint then corrective action may be necessary. If the register is in the read set, the successor is squashed and restarted, as sequential semantics were violated. Otherwise, if it is in the spill set, then we fix up register and memory state. If it is not in the spill set, and it is in the write set, then no action is required. Finally, if it is not in any of the register sets, then the register state is fixed (so that it can be correctly forwarded to the next epoch). For all live spill locations, the memory value in the SSB is updated before writeback can occur (if the register starting value has changed), thus restoring sequential semantics.

Limitations The approach above can generally track spills, with two main limitations. Firstly, in the hardware implementation, the register-address store has limited size. If this overflows, the microarchitecture can squash the youngest epoch, freeing up all of its entries. This eventually frees up enough entries, as the oldest (architectural) epoch does not need entries. In case the youngest epoch is the one committing the spill instruction, it can be temporarily stalled instead of squashing it until sufficient entries are available.

Secondly, stores that partially overwrite a spill location are tricky to handle, since a part of the spilled value will still remain in memory. It is possible to handle this by adding a byte-level bitmask to each spill location (and only considering loads from fully un-masked locations as fills). A simpler approach is to consider the value as observed as soon as a live spill location is partially over-written, which should have only a small impact except in pathological cases.

3.12 Evaluation

Now that I have shown how to design, construct and target a system to perform in-core, hint-based, speculative multithreading, I will evaluate the basic design. I show that it can

[15] This can be limited to only callee-saved registers under the architecture's common calling convention.

achieve 6.4% speedup over a sequential (out-of-order superscalar) baseline, and where these benefits come from. I also analyse the inefficiencies present in the current design, which will be improved later in the thesis.

3.12.1 Simulation methodology

We obtain the data presented in this section following the *SimPoint* [29] methodology to select and simulate representative regions of whole-benchmark runs of SPEC CPU2006 [33] workloads with full-sized 'reference' inputs. To simulate perfect static region selection, we run each SimPoint multiple times, to measure the possible speedups from each promising unique region. From this, a list of loops is selected to maximise speedups, and this is used in a final timing run (of all simpoints, with the loop list) to measure overall performance.

I have implemented the timing model in Gem5, and performed the analysis of results, but the other tools used here (i.e. compiler, functional simulator, loop selector) were largely developed by my group and collaborators from Arm.

Compilation

We produce AArch64 binaries using the compiler toolchain shown in Section 3.3, in which most loops are transformed and annotated with parallelisation hints. Only a small number of loops are excluded, either because they yield completely empty parallel bodies, or the compiler is unable to canonicalise them, and so our (loop) pass cannot analyse and transform them. The binaries are optimised with the `-O3` option, with all optimisation passes (including vectorisation) enabled. The loop annotation pass runs last in the middle-end pipeline, on fully optimised binaries.

We measure the performance degradation from hint insertion on a real system[16] by comparing results between a version compiled with an unmodified version of the LLVM compiler toolchain, and a version obtained by manually replacing any hints with `NOP` instructions in the assembly code produced by our compiler. The geometric mean performance degradation is 0.2%, with a maximum of 2.3% (for *omnetpp*), and all other benchmarks slowing down by 1% or less.

This is not surprising, since `NOP` instructions are extremely cheap on modern, wide superscalar machines. Furthermore, since most of the annotated loops will not be used for parallelisation, it is reasonable to assume that a production-quality compiler could filter out the vast majority, and thus produce negligible slowdowns from hint insertion itself. Specifically, filtering out inner loops that only have a handful of instructions in the body should mostly achieve this, since the frequency of hints is very low in larger loops. Thus, we ignore the slowdowns from hint insertions, and use the annotated binary as a baseline for evaluating the microarchitecture.

Functional simulation

With the annotated binaries, we first use a functional simulator to perform full (start-to-finish) runs of SPEC CPU2006 workloads, with reference inputs. The functional simulator runs

[16] Intel® Xeon® W-2195 processor (4-wide OoO superscalar, released in 2017).

all user-mode instructions, and it emulates system calls. At the end of the run, apart from outputting some basic data about parallel regions encountered (including the number of dynamic instructions inside the region, and the number of matching hints executed, by hint type), it also prints out *basic block vectors* and their frequencies, to be consumed by the *SimPoint* [29] tool. Once the selected SimPoints and their weights are obtained, we invoke the functional simulator once again. This time, it pauses a set 'warmup' offset before each identified SimPoint region (identified by a dynamic instruction offset from the start of the run), it captures architectural state (i.e. architectural register, memory and operating system state), and packages these up into a checkpoint. The simulator produces a checkpoint for each selected SimPoint. The output from this tool is a checkpoint and associated representative weight for each selected SimPoint. The weights add up to 1 for each workload, and they signify the fraction of dynamic instructions in the full run that are represented by each SimPoint.

Timing simulation

We implement a detailed pipeline and cache model in Gem5 [5, 48] for timing results. The idea is to invoke this model twice per SimPoint – once with parallelisation disabled, and once with it enabled – in order to calculate speedups and changes in other metrics. In both modes, Gem5 uses the corresponding checkpoint to initialise its architectural state. The initial warmup period lasts for a set number of instructions, and it is included to ensure that the system (specifically, caches, buffers, prefetchers and predictors) reach a steady state, closely mirroring normal, back-to-back execution of SimPoints. Threadlet speculation is active during the warmup, in order to warm up speculative state correctly. After warmup is done, the simulator starts timing and statistics collection, and it runs until a fixed number of dynamic instructions commit to architectural state. In the sequential version, this can happen in any cycle when instructions pass the *commit* pipeline stage, while in the speculative version, it can happen either when an epoch commits to architectural state, or in any cycle when instructions from the architectural threadlet pass *commit* in the pipeline.

Loop selection

Since we did not explore loop selection in the compiler or the microarchitectural prototype, we need to help Gem5 to select profitable loops. Since loops may overlap (in the case of nested loops or function calls inside loops), we need to not only select loops that are profitable, but find the most profitable set. For example, if two nested loops are both profitable, but the inner loop gets more speedup, then we are clearly better off disabling parallelisation on the outer loop, and targeting the inner loop instead. In the limit, each dynamic region encountered (or even, each iteration) would be individually parallelised or ignored.

We acknowledge that this is a complex design space, and future exploration is needed. For this evaluation, we assume that the compiler can select the most profitable (static) loop addresses correctly, through a combination of heuristics, cost models and profiling. In reality, we expect the microarchitecture to need to add run-time knowledge to facilitate this, but this space is also left to future work.

To simulate perfect static loop selection, we run most[17] loop regions individually in each SimPoint they appear in, by instructing Gem5 to ignore all other hints. This run outputs the single-region speedup per region per SimPoint, as well as an activity bitmask. The bitmask divides the dynamic instruction stream of the SimPoint region into 64 buckets (for ease of computation in our script), and sets each bit if the chosen region was active in the corresponding part of the run.[18]

We then perform backtracking search (with pruning for efficiency) to find the most profitable combination of regions across the whole workload. To do this, we pessimistically assume that two regions that were both active in a given bucket produce the worse speedup out of the two for that bucket. Furthermore, we assume that the benefit (i.e. cycles saved) from a given region in a given SimPoint come uniformly from the individual buckets that region exhibited activity in. That is, the combined cycle savings S_{AB} from taking both regions A and B in a SimPoint are as follows:

$$S_{AB} = S_A \cdot \frac{C_A}{C_A + C_{AB}} + S_B \cdot \frac{C_B}{C_B + C_{AB}} + \min\left(S_A \cdot \frac{C_{AB}}{C_A + C_{AB}},\ S_B \cdot \frac{C_{AB}}{C_B + C_{AB}}\right), \tag{3.1}$$

where S_A and S_B are the cycle savings from only A and only B, respectively, C_A and C_B are the number of buckets that either A or B were active in, but not both, and C_{AB} is the number of buckets where both regions were active. The first term is the savings from A in the C_A buckets where only A was active, the second term is the savings from B in its C_B unique buckets, and the third term takes the minimum savings from either A or B in the C_{AB} shared buckets where they may have been nested. This approximates the total benefit, as only one region (the outer one) will be active at a time. Under-estimating here makes sense, as we prefer making a concious choice between two mostly-overlapping regions, based on the speedup numbers, rather than leaving it up to chance at run-time (i.e. choosing whichever region is the outer one).

The benefits from an arbitrary set of regions is obtained similarly, by repeatedly applying this formula and combining the activity masks (using bitwise-or). Using these partial values, we approximate the overall benefit across all SimPoints by summing up the cycle savings from the current set across all SimPoints, weighed by the SimPoint weights. From these approximations, the best set of loops is selected, and output into a loop list for the final run.

Final Timing Run

We save the selected list of loops into a file, and re-run the full timing simulation on each simpoint with the loop list as an input, in order to get a more accurate final speedup number, without the approximations described above. The loop list is the same for every simpoint in a workload.

[17] We apply minimal pre-filtering in order to reduce the number of experiments. See details in table 3.1. This filters out loops that are unlikely to be profitable or have a significant impact.

[18] The activity bitmask is an approximate way to estimate the overlap between different regions. This simple method suffices, as the main goal is to select only the best loop in each loop nest, instead of all profitable loops, as the microarchitecture will only parallelise one loop at a time.

Results

We calculate the final results from the statistics and timings of the two final Gem5 runs (parallel and sequential) of the same SimPoint. Full-workload results are inferred from the per-SimPoint results, SimPoint weights, and the total number of instructions in the workload. For correctness [29], I take care to calculate the numerator and denominator of complex metrics separately, and only divide the final values. This includes metrics based on fractions of time (e.g. Figure 3.15), and speedup. Thus, speedup is calculated as parallel total time divided by sequential total time.

Some SPEC benchmarks contain multiple workloads (one per input), of different sizes. For these, I sum the run times across the different invocations, following how SPEC's default toolchain operates.

3.12.2 System parameters

Table 3.1 shows the simulation parameters. I simulate an aggressive, 8-wide out-of-order core in Gem5, with the best available prefetchers and branch predictor. Choosing a setup that is wide and deep enough is important for results that are applicable to modern industrial designs. We chose not to scale to a larger design, as Gem5's simple, weak front-end design and somewhat outdated predictors make such large systems unrealistic.

3.12.3 Speedups

Figure 3.14 shows the speedups of the parallel system over a sequential baseline with the same pipeline parameters over the full SPEC CPU2006 benchmark suite. We observe measurable (above 1%) speedups for 12 out of the 22 benchmarks, as highlighted on the figure. The remaining plots in this section only show these benchmarks, ordered by speedup from best to worst.

The best whole-benchmark speedup is 34%, achieved on the *calculix* benchmark, followed by 15% or more on *bwaves*, *milc*, *zeusmp* and *omnetpp*, and small, but measurable gains on 7 other benchmarks. Most other benchmarks have a handful of loops that are either

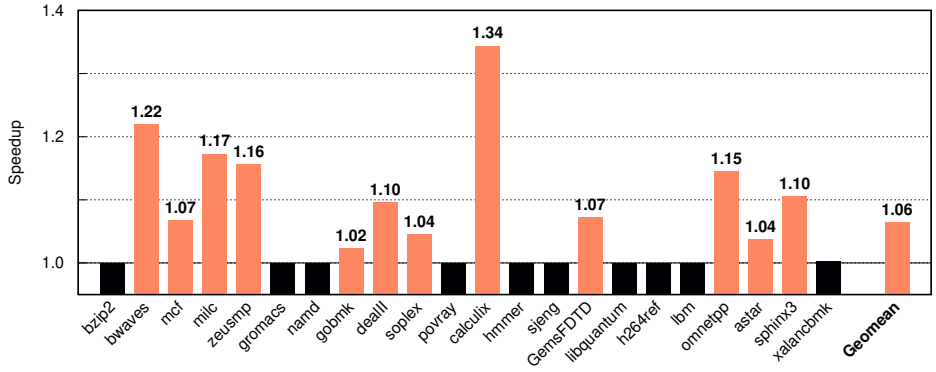


Figure 3.14: Whole-benchmark speedup over the single-threaded out-of-order baseline, running the SPEC CPU2006 benchmark suite (SimPoints from full 'ref' inputs). We observe a geometric mean speedup of 6.4%. Benchmarks with 1% speedup or more are highlighted.

SimPoints & Filtering	
SimPoints	250 million instructions per SimPoint, 50 million instructions warmup
Loop filtering	$\leq 10^5$ instructions per iteration, ≥ 1.5 iterations per invocation, ≥ 200 instructions per invocation on average, and $\geq 0.1\%$ dynamic instruction coverage per SimPoint
Core	
Pipeline	4 GHz, 8-wide, dynamically shared between threadlets
Threadlets	4 threadlet contexts
Fetch	4 fetch buffers (partitioned 1×4 / 2×2 / 4×1), max 4 insts per buffer, picking at most 8 instructions/cycle/threadlet, 64-entry Fetch Queue per threadlet
Structures	1024-entry ROB, 256-entry IQ, 64-entry LQ and SQ
Register File	Dominated by ROB (1280 physical int registers)
Execute Pipes	7 ALU+Branch, 2 ALU+Mul+Div, 4 SIMD+FP (2 Div/Sqrt/SimdMul + 2 SimdShift/Cvt), 4 Load pipes, 2 Store pipes
Branch Predictor	256 Kbits LTAGE [72] (13-component TAGE + 256-entry Loop), 4096-entry BTB, 48-entry RAS. All contexts share one predictor, with separate histories.
SSB	
Work Queue	32-entry (including in-flight architectural ops and reattach ops)
Granule Cache	8 KiB per slice (32 KiB total), 1 B granules
Latency	Reads: 0 cycles (parallel with L1D lookup), speculative writes: 1 cycle
Conflict Detector	Byte-level granules, unlimited read set
Memory System	
L1I	64 KiB 4-way, 1-cycle hit, 16 miss-state handling registers (MSHRs) ($\times 8$ targets)
L1D	64 KiB 4-way, 2-cycle hit, 10 MSHRs ($\times 16$ targets), 12 write buffers, stride (deg: 2) prefetcher
L2	4 MiB 8-way, 11-cycle hit, 32 MSHRs ($\times 16$), 32 WBs, tagged, stride (deg: 8) and neighbor prefetchers
DRAM	32 GiB DDR3-1600 memory, $\sim$ 35 ns (140-cycle) round-trip access latency, including caches

Table 3.1: Simulation parameters.

only marginally profitable, very small, or both, and so they do not produce whole-benchmark gains distinguishable from noise, and so most of them are thus not selected (only *bzip2* and *xalancbmk* had between 0% and 1% speedup).

Whole-benchmark speedup depends on the speedups obtained on individual regions, the size of those regions, and any changes in performance outside of regions. Outside of regions, we do not observe measurable changes in performance.[19] The region selection identifies 79 profitable (static) regions, yielding 3 million dynamic region invocations, of which 2.3 million are profitable and 0.7 million produce a slowdown[20]. The geometric mean speedup achieved by static regions is 23.8%, with the best region reaching 3.0× speedup, and 53 regions yielding at least a 10% improvement.

Although this is a promising start, naturally there are inefficiencies that we can improve on. The remainder of this evaluation aims to give an overview of where the speedups come from, identify bottlenecks to target in the next chapter.

3.12.4 Speculation

Figure 3.15 shows the amount of speculative activity, measuring the proportion of time when 1, 2, 3 and all 4 threadlet contexts were active, respectively. The vast majority of time (26.8% out of 28.2%) with only 1 threadlet active is spent outside parallel regions. On the profitable benchmarks, this translates to at least two threadlets running 77% of the time. Compared to this, however, all four threadlets are only active 44% of the time. This suggests that finding and efficiently mapping enough parallel epochs is a main bottleneck. Furthermore, this plot also shows that over the whole benchmark set, we only perform speculation in 26.8% of the run time (and full, 4-threadlet speculation in only 16.2%). Using Amdahl's Law [2] and assuming a maximum of 4× in-region speedup (from 4 threadlets) on the targeted 26.8%,

[19] Only four out of the nearly 600 total SimPoints showed more than 0.5% reduction in the time spent outside regions compared to the baseline (maximum: 2.7%). This is likely due to incidental perturbations, and these SimPoints did not have particularly high speedups.

[20] These are from static regions that have both profitable and unprofitable invocations, but are overall profitable.

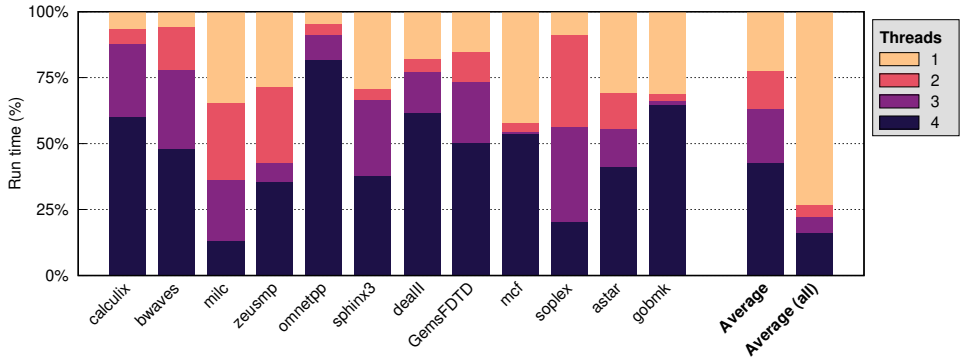


Figure 3.15: Percentage of run time with N active threadlets. $N = 1$ means that only the architectural threadlet is active, $N = 2$ means 1 active speculative threadlet, etc. The average coverage across all 22 benchmarks is 16.2% with all 4 threadlets active, and 26.8% with at least 2 active threadlets.

we can derive that the upper bound on the maximum achievable whole-benchmark speedup without expanding coverage is 25%. With a more common 2.0× maximum speedup, this bound falls to 15%. Therefore – apart from improving speedups on the current loops – we should also aim to improve the techniques in a way that allows more loops to be profitably parallelised. This will be addressed in the next chapter.

3.12.5 Squash rates

Figure 3.16 shows the fraction of speculative epochs that get squashed due to true cross-epoch memory dependences.

For the two outliers – *omnetpp* and *mcf* – all speculative epochs fail due to a conflict, but they effectively prefetch lines into the cache, and this side-effect causes their architectural (re-)executions to complete faster due to fewer cache misses.

For the remainder of the workloads, squash rates are lower. The squash rate for *astar* is 20%, for *GemsFDTD* it is 14%, *calculix* squashes 10% of epochs, and all remaining workloads exhibit less than 10% of speculative epochs failing due to conflicts. At first glance, this seems to suggest that most (80% or more) loop iterations are independent in the benchmarks tested, but it is important to note that these squash rates suffer from selection bias. We simulated perfect static loop selection for a specific parallelisation scheme, and discarded all loops that did not perform well. Therefore, we can only conclude that (apart from the outliers, *omnetpp* and *mcf*) we only found loops with relatively low conflict rates to be profitable. Indeed, many of the loops not chosen by our loop selection process did observe high conflict rates, but they were not (sufficiently) profitable to be included in the optimal selection. Improving performance for these loops will be addressed in the next two chapters.

Apart from conflicts (shown in Figure 3.16), squashing can also occur due to the region reaching a sync instruction in a predecessor, and the SSB slice filling up. Squashing on a sync only occurs in *calculix* and *gobmk*, where we have early exits from the body. This only accounts for an insignificant number of squashes (0.4% and 0.0%, respectively) in these benchmarks. Squashes due to the SSB slice being full occur in multiple workloads, but

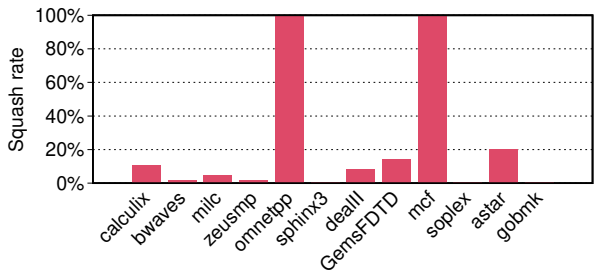


Figure 3.16: Squash rate of speculative epochs due to cross-epoch conflicts. 0% means all speculative epochs that reach conflict checking commit successfully, 100% means all of them fail.

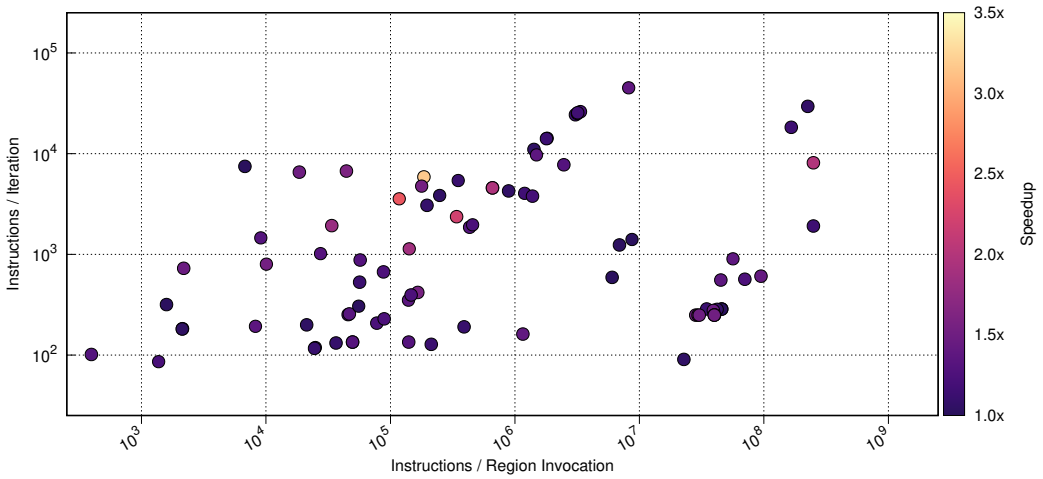


Figure 3.17: Profitable parallel regions based on the dynamic size per iteration and invocation, coloured by speedup.

only *zeusmp* and *mcf* exhibit a significant fraction of these squashes. For *zeusmp*, 71% of squashes are due to an over-full SSB, and this number is 50% for *mcf*.[21]

As seen here, squashing due to a full SSB does not significantly alter performance in most cases. Indeed, the overall geomean speedup with an unrestricted SSB is 6.7% (as opposed to 6.4% with the SSB restricted to storing 8 KiB for each of the 4 epochs). In the case of *zeusmp*, restricting the SSB hurts performance (speedup reduced from 25.2% to 15.6%). For other benchmarks, the restriction has little impact on performance.[22]

3.12.6 Parallel regions

Figure 3.17 shows the individual profitable (static) parallel regions. The iteration sizes range from just 86 instructions to $6.2 \cdot 10^4$ instructions, while region sizes range from under 395 to $2.5 \cdot 10^8$ instructions. While these are both wide ranges, most of the more profitable regions sit around the middle. It is interesting to observe the causes for each of the four bounds.

We expect regions with too many instructions per iteration will not be beneficial, because their working sets are likely to be too large. This causes issues for two reasons. Firstly, it is unlikely that the SSB can store all the state, and therefore epochs will abort due to exceeding size limitations, before they can commit any state. Secondly, with increased delays and increased working sets, useful prefetching side-effects gradually turn into cache pollution, as the prefetch is too early. Lines brought in by the speculative threadlet will even be evicted before being used after a few tens of thousands of instructions.[23]

[21] Note that with the current experimental design, epochs that fill their SSB slice squash and restart. This means they may later squash again (any number of times) due to re-filling the SSB slice, and they could squash once again due to conflicts detected at commit time.

[22] Although we did not manage to correctly simulate *mcf* with an unrestricted SSB, due to failures caused by the extremely high amount of speculative state

[23] Our 64 KiB first-level data cache has 1024 lines (64 B each). Assuming a memory access every 5 instructions, and a 5% miss rate (which is generously low for applications targeted by prefetching), we have a cache miss every 100 instructions. Thus, the average cache line will be evicted after $100 \times 1024 \approx 100K$ instructions. This reduces to $50K$ instructions, if we assume half the cache is taken up by frequently accessed lines, such as those

There is no inherent reason for very long regions to perform badly. Indeed, some very large regions produce good speedups (e.g. in *omnetpp*, as shown in Figure 3.18(b)). On the other hand, there are more possible values for the size of iterations for these loops, and so it is more common for long-running loops to have large iterations, which are tricky to parallelise.

Regions whose iterations are too small – on the other hand – suffer from threading overheads. Even though detaching and reattaching threadlets is very cheap, it is not free. Filling up and emptying the out-of-order windows of the threadlets takes a couple of tens of cycles, and this overhead reduces speedups and can even lead to slowdowns. Tackling some of these regions is explored in Section 4.3.

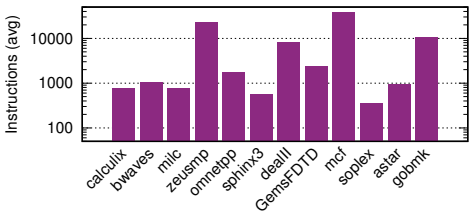
Similarly, regions that are too small cannot be profitable, because of overheads. In particular, regions with few instructions per invocation either have very few iterations to reach steady state and overlap significant amounts of computations, or their iterations are too small (as discussed above), or both.

3.12.7 Epochs

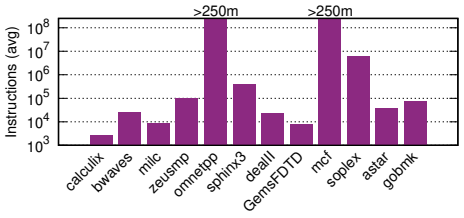
Figure 3.18(a) shows the average size of epochs in each benchmark, measured in dynamic instructions. We can see that this number is similar for most benchmarks, just below a thousand instructions, with some benchmarks featuring longer epochs. This confirms the fact that most profitable loops are in the 10^3 to 10^4 instructions per iteration range, meaning that profitably covering loops outside of this region is a promising way to expand coverage, and increase whole-benchmark speedups. For covering loops with smaller bodies, we need to make sure we can efficiently utilise threadlets even when the loop iterations are short-lived (as discussed for Figure 3.15). As we can see from the outliers, some larger loops can be covered without issues, but we cannot cover loops with cross-iteration conflicts (as shown in Figure 3.16). Since larger loops are more likely to result in conflicts, reducing cross-iteration conflicts as much as possible should help us cover more of these loops. I investigate solutions to both of these issues in the next chapter (see Sections 4.2 and 4.3).

Figure 3.18(b) shows the average size of parallelised regions (loops) per benchmark. As expected, this exhibits a very high degree of variance between workloads. While very

storing the stack, globals and other 'hot' data structures. These lines are not candidates for prefetching, as they already hit due to temporal locality.



(a) Average size of epochs in dynamic instructions.



(b) Average size of selected parallel regions in dynamic instructions.

Figure 3.18: Speculation epochs across the SPEC CPU2006 suite. These figures are averaged across all parallel regions in the benchmark.

small regions are unlikely to be beneficial (due to constant costs), long-lived regions have no inherent reason to be unprofitable, which explains the wide range of values. In *omnetpp* and *mcf*, regions could not be measured accurately, as they are larger than the SimPoint interval of 250 million instructions.

3.12.8 Case studies

I performed a deeper manual analysis of the top loops in benchmarks that achieved above 10.0% speedup, using detailed simulator performance statistics and timing traces, and present the results here.

Calculix The top loop here features a long dependence chain made up of complex floating-point and vector instructions. This leads to approximately 40% of cycles seeing no instructions being issued. Sharing the ROB between 4 contexts hurts the performance of the oldest threadlet, reducing IPC from 1.6 to 1.0. However, we get an additional 0.9 and 0.6 IPC (respectively) from the first two speculative threadlets. Most of the additional instructions are issued in cycles when the original threadlet cannot find any instructions to issue. With an observed loop size of 725 instructions, the baseline microarchitecture barely saw past one iteration (with 1024 ROB entries). In contrast, sharing the ROB entries, the new threadlets speculate approximately 1000 and 1700 instructions into the future, and effectively reveal three independent chains of operations. Accounting for a slight squash rate, we still get a healthy speedup. The youngest threadlet is not utilised well, likely due to the first three contexts saturating floating point pipe bandwidth.

Bwaves Similarly, baseline IPC in the main loop is 1.2, with the addition of speculative threadlets reducing IPC in the main threadlet to 0.8, but speculative threadlets recouping the loss (adding 0.5, 0.2 and 0.1 IPC, respectively) due to increased look-ahead and the resulting additional memory-level parallelism. Curiously, cache miss rates and cycles when the cache is blocked increase, but memory read pipes are less contested. Presumably, high-latency memory load instructions are discovered at a more steady rate, rather than in bursts (due to having multiple streams of instructions), leading to less peak contention and a more balanced mix of different instruction types.

Milc Most of the speedup is obtained from 5-10 lower coverage loops, with similar characteristics. IPC is abysmal (close to 0.5), due to memory latency. Threadlets increase look-ahead, thus exposing extra memory-level parallelism, and distributing the discovery of load instructions more evenly, leading to better utilisation of read pipes.

Zeusmp Similarly to milc, zeusmp is memory-bound, and adding multiple threadlets increases memory-level parallelism and reduces read pipe contention. However, loops here are much larger (up to 6000 instructions per iteration), and squash rates are higher in the top loop. Instead, the prefetching effect of speculative threadlets pushes up the oldest threadlet's performance. When a speculative threadlet reads a cache line it is brought into the first-level data cache (L1). Even if the speculative threadlet is squashed, when the architectural

threadlet re-executes the access, it will now be an L1 hit (unless it got evicted). In zeusmp, this phenomenon is especially clear, with the bottleneck shifting from L2 cache accesses in the baseline to L1 accesses in the parallel version's architectural threadlet.

Omnetpp As discussed above, all speculation is squashed in omnetpp, but we still get a speedup from the prefetching effects. The algorithm in this benchmark pops work-items from a priority queue in the header section of each iteration, and then processes them in the parallel body. As the data structure is modified elsewhere in the body – which also triggers a re-balancing of the binary tree representation – conflicts are detected on the tree walks and updates. However, the speculative work used to determine what to add or modify in the tree is largely unaffected by these conflicts, thus doing it speculatively acts as a really sophisticated prefetcher for the architectural threadlet, improving its performance.

Sphinx3 This benchmark relies on two main loops, covering about two thirds of the benchmark, and exhibiting similar behaviour. The baseline code has poor performance due to frequent branch mispredicts (>1.5 MPKI), most of which are resolved late due to long-latency dependence chains. As a consequence, the later entries in the re-order buffer are either empty (because the front-end has not caught up yet) or they contain instructions that will be squashed once the misprediction is discovered. In either case, the entries are typically wasted, thus adding 3 more threadlets (and sharing ROB entries between them) does not noticeably reduce the speed of the architectural threadlet in this case. Even though the speculative threadlets collectively only commit 15-20% of instructions (due to lower priority), this yields an approximately 20% uplift in IPC.

In summary, speedups come from different sources across different benchmarks and loops. These include primary effects, such as increased look-ahead exposing multiple independent chains of instructions, higher levels of memory-level parallelism, and limiting the impact of branch mispredicts, as well as secondary effects such as prefetching, a less bursty supply of instructions, and a more constant mixture of different instruction types.

3.12.9 Area and Power Overheads

The main overheads are the addition of threadlets to the pipeline, the SSB, and the conflict detection logic.

Handling threadlets is a simplified form of SMT, which requires a 10–15% increase in area and power [9, 19, 46]. The SSB's main storage area is the granule caches, consisting of 4 slices with 8 KiB capacity each, corresponding to an approximate area of 0.10 mm^2 per slice (0.03 nJ per access) according to CACTI [54] (22 nm, `itrs-hp`, 4-way associative, 1 read/write port, 1 read-exclusive port per bank). In a 7 nm node, using a conservative scaling factor of 5, the area is 0.08 mm^2 for all four slices. Finally, an implementation of conflict checking may be based on bloom filters, similar to Swarm [37], taking up approximately 0.005 mm^2 of area at 7 nm (dual-ported SRAM with 8 entries, and 4,096-bit filters). Against the total area of a relatively high-end core, e.g., an Arm Neoverse N1 (1.4 mm^2 at 7 nm, including private L1 and 1 MB L2 caches [62]), the area of these additional

components is around 6%. Hence, we expect a total increase of 16–21% in area compared to a sequential design, making the 6.4% headline speedup figure slightly below the 8–10% expected according to Pollack's rule [7] (although its validity for today's cores is debatable).

3.12.10 Sensitivity to parameters

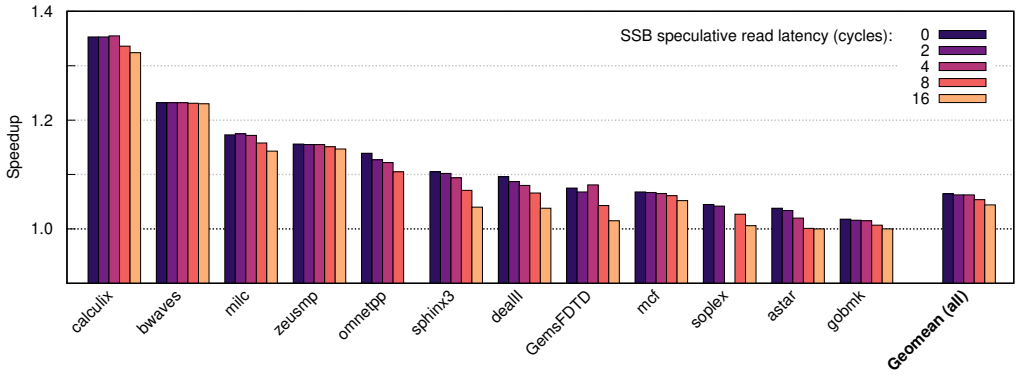
Figure 3.19 shows the sensitivity of the speedup results to SSB and conflict checker latency parameters. As expected, higher latencies result in reduced speedups, as the added latency affects the parallel version, but not the sequential baseline. However, notably, the speedup figures are relatively stable for many benchmarks, even in the face of relatively high latencies.

Figure 3.19(a) shows that effects of adding read latency to the SSB. The SSB read lookup is performed for speculative reads, and it can be started in parallel with the first-level data cache (L1D) access. Thus, the latency for providing the data here is in addition to the 2-cycle hit latency of the L1D cache, and likely accounts mainly for patching together the values coming from the SSB and the data cache. Since this is similar to operations implemented in store buffers in a single cycle, the fact that two cycles of added latency here only brings the geometric mean speedup down from 6.4% to 6.3% is promising. Furthermore, even 4 cycles of added latency only reduce the speedup to 6.2%, making sequential lookups with the data cache acceptable. Higher latencies start having larger effects with the geometric mean reducing to 5.4% for 8-cycle accesses and to 4.4% for 16 cycles, since the out-of-order pipeline starts to struggle to mask delays. Note that when setting read latency to N cycles, I also included an N-cycle delay before epoch commit, in order to simulate the conflict detector checking these last read accesses after they finished.

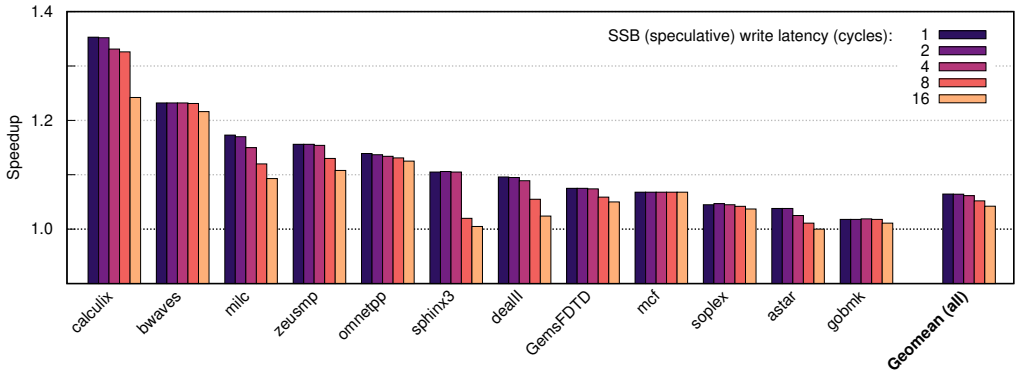
Figure 3.19(b) shows a similar story for speculative write latency. Here, the base case has a 1-cycle latency, as writes do not have to wait for accesses to the data cache to finish. Speedups are virtually unaffected if the delay is increased to 2 cycles (6.4%). Performance declines slightly, to 6.2% with a 4-cycle latency, and it declines more sharply for higher latency values, with speedups falling to 5.2% with 8-cycle writes, and 4.2% when writes take 16 cycles.

Figure 3.19(c) shows the effects of adding latency before epoch commit. This accounts for conflict checking to run (or finish in case it is performed in the background). Speedups are more resilient to latency here compared to increasing read or write latencies, as this delay is only encountered once per epoch (as opposed to once per memory access). On the other hand, these delays are masked less well by instruction-level parallelism, since an entire threadlet context is standing idle and empty until the commit can finish (as it may still need to be restarted in case of a conflict). The speedup only drops to 6.0% with a 16-cycle commit delay, 5.4% at 32 cycles and we still get a 4.8% speedup with 64 cycles of delay per commit. We can see that the drop in speedup comes from benchmarks with short epochs (see Figure 3.18(a)). The other predictors of loss of performance are instructions per cycle (IPC) – since delays are less frequent if executing each epoch takes longer – and conflict rates, since I only add the pre-commit latency to successful epochs.[24]

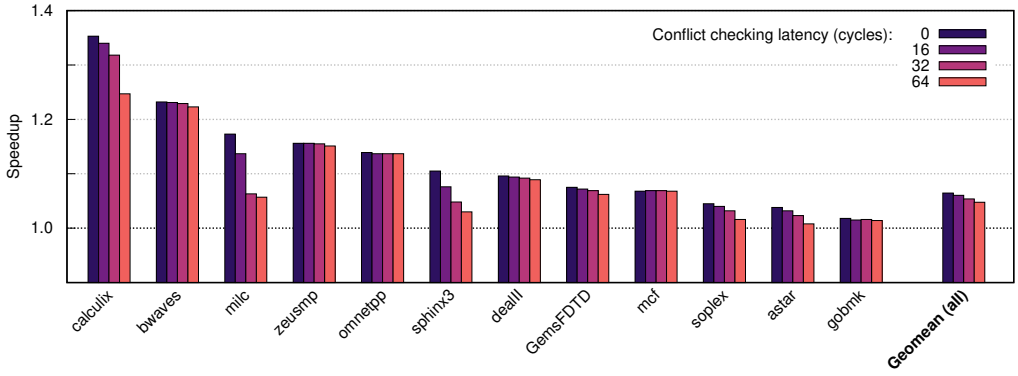
[24] I make this assumption, as I assume that conflict checking runs in the background (although this is not modelled by the simulator), and so – in most cases – the checker would already see conflicts early, but it would need extra cycles to finish checking the latest accesses.



(a) Sensitivity to speculative read latency in the SSB. Latency is added to all speculative reads (in addition to the level-one cache latency). Two simulation failures (for *omnetpp* and *soplex*) are omitted.



(b) Sensitivity to write latency in the SSB. Latency is added to all speculative writes.



(c) Sensitivity to conflict checking latency before epoch commit.

Figure 3.19: Sensitivity of speedups to changing the latency of the SSB and conflict checker. Only benchmarks with 1% speedup or more are shown, but the geometric mean considers all benchmarks.

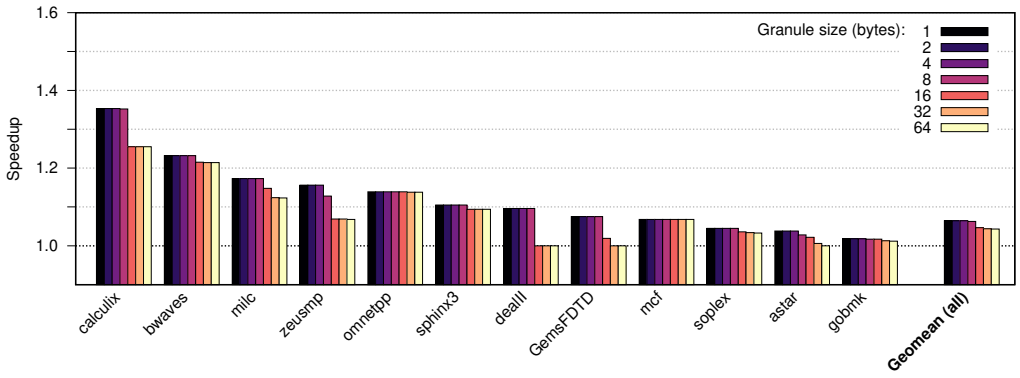


Figure 3.20: Sensitivity to conflict checking granule size. For larger granule sizes, neighbouring addresses on the same (address-aligned) granule exhibit false aliasing.

Figure 3.20 shows the sensitivity of speedups to the granule size used in the conflict checker. With a granule size of 2^B (bytes), the B least-significant bits of each granule address are zero. For example, we would have 8-byte granules at `0x3FDC40`, `0x3FDC48`, `0x3FDC50`, `0x3FDC58`, etc. When an access is made, conflict checking is performed for each granule it (partially or fully) accessed. For example, with an 8-byte granule size, the 16-byte (`0x10` bytes in hexadecimal notation) read access (starting) at address `0x3FDC44` would be seen by the conflict detector as a read from the granules `0x3FDC40`, `0x3FDC48` and `0x3FDC50`, as it affects the last 4 bytes of `0x3FDC40`, all of `0x3FDC48`, and the first 4 bytes of `0x3FDC50`. Then, (write) accesses from prior epochs that overlap (partially or fully) with any of these three granules would show up as a conflict. For example, the 4-byte write to `0x3FDC40` would trigger a conflict, even though it does not overlap with the read in memory at all. I assume that the SSB can still use byte-level masking, in order to implement partial writes. An alternative would be to treat partial writes as a read followed by a write.

We can see that speedups are virtually unchanged for granule sizes 1, 2, 4 and 8. This is not entirely surprising, as the word size of the machine is 8 bytes, therefore on full-word accesses (e.g. arrays of 64-bit integers or double floating-point values), these granule sizes would not introduce any false aliasing. A limited number of accesses are made at smaller granularities, but this only has a limited impact on performance. With a granule size of 8, we lose 2.8% of performance for *zeusmp*, and 1.0% for *astar*, resulting in a 0.2 percentage point geometric mean regression only. For larger granule sizes, aliasing starts to be a problem, reducing the initial 6.4% geometric mean speedup to 4.7% with 16-byte granules, 4.4% with 32-byte granules, and 4.3% with 64-byte (or cacheline-sized) granules due to the increased squashing. Speedups completely disappear for *dealII*, *GemsFDTD* and *astar*. This suggests that conflict checking at word (8-byte) granularity is the optimal choice.

3.12.11 Summary

This section evaluated the performance of the fully-functional base design for in-core, hint-based, speculative multithreading presented in this chapter. The design achieves 6.4% geometric mean speedup (see Section 3.12.3) using approximately 16–21% area overhead in a naive design (see Section 3.12.9). It exhibits low sensitivity to small changes in granule

size, SSB access latency and conflict checking latency (see Section 3.12.10), although losses increase quickly past a certain threshold.

The speedup is obtained by parallel threadlets running during only 27% of the run time, and all four threadlet contexts being utilised only 16% of the time in total, as shown in Section 3.12.4. Utilisation is far higher (77% for some, and 46% for all threadlets) in the profitable subset of benchmarks. This suggests that low profitable coverage and threadlet utilisation are major bottlenecks, as even assuming static 2× speedup, 27% coverage can only yield 15% overall speedup. Furthermore, Section 3.12.5 found that all but two profitable benchmarks have relatively low conflict rates due to selection bias, and many unprofitable (and thus not selected) regions show higher conflict rates. Finally, Section 3.12.6 showed that loops with small iterations tend to get get lower or no speedups. These issues will be addressed in the next chapter.

Chapter 4

Optimisations

In the last chapter, I presented a functionally correct in-core, hint-based, thread-level speculation scheme. As shown in the evaluation (Section 3.12), this scheme achieved some speedups, but many regions could not be profitably covered, due to performance issues including squashes, low context utilisation and performance for small iterations (as summarised in Section 3.12.11). To tackle these performance issues, this section introduces two important optimisations. This is followed by an investigation of the resulting performance and remaining bottlenecks.

4.1 Summary of optimisations

The first optimisation – *eager memory value forwarding* – reduces conflicts by making values produced in one epoch available to future epochs early (when they are stored to the SSB), as expanded in Section 4.2. This means that younger epochs can only observe stale values if a true read-after-write dependence exists between the epochs *and* the SSB performs the write operation after the read. Since this eager value forwarding reduces the number of times stale values are read, we can relax and expand the conflict detector's logic to distinguish between read-after-write dependences where the correct ordering has or has not been respected, and only trigger a squash if ordering may have been violated (the checking uses some approximation, which is discussed in more detail in Section 4.2.3).

The second optimisation – *iteration packing* – targets loops with small iterations. If possible, the microarchitecture predicts the iteration size in (dynamic) instructions, and the evolution of (input) register values between iterations. Provided it can make predictions with high enough confidence, it groups ('packs') together multiple iterations into each epoch, in order to increase the epoch size. Doing so helps to amortise the static per-epoch overheads, and improves the utilisation of threadlets, resulting in better performance. This optimisation is discussed in Section 4.3.

I also implemented a number of other optimisations, but these do not bring a benefit on the SPEC CPU2006 benchmark set used for evaluation after applying the above two optimisations. I will discuss these in Section 4.5, after investigating the two impactful optimisations in detail first.

4.2 Eager forwarding between epochs

The implementation of the speculative state buffer and conflict detector, as presented in the previous chapter, can lead to a conflict between a write and a read (from different epochs), even if the write happened first. In fact, a conflict is found every time between a write W from epoch E_W and a later read R from epoch E_R if $E_W < E_R$. Even though these are true read-after-write hazards, they do not need to result in a conflict, as the written value is produced before the read is satisfied from the SSB. However, since the written value is stored in the slice of its epoch, and this is different from the read's epoch, the value is not yet available to be consumed by that read in the earlier SSB design.

This section presents a more sophisticated SSB design, which is capable of forwarding such values, thus eliminating all remaining hazards except true read-after-write hazards where the write happened after the read in the SSB.

Apart from reads and writes that probabilistically happen in the correct order, value forwarding also eliminates read-after-write conflicts from writes in the header. Additionally, it also enables eager conflict checking and early restarts (as described in Section 4.2.3), since conflicts can no longer happen between writes that happened before the read's epoch was restarted.

4.2.1 Unified speculative state buffer

Let us logically view the different slices of the SSB as a single, unified cache. Each granule in this cache is addressed by a pair of values, consisting of the memory address and the epoch ID. This design is capable of operating in the same way as the earlier one, but we can add the desired forwarding behaviour, as all the necessary values to serve a read request with the most up-to-date data (whether from the same epoch or any past epochs) are available in a single cache. The actual structure of the SSB remains the same as shown in Figure 3.10, but we alter the access logic to allow for eager forwarding of memory values, as described below.

Figure 4.1 shows how such a most up-to-date version can be identified for a load, taking values from the memory system, and any past (architectural or speculative) epochs, as appropriate, but ignoring future epochs (depending on the epoch the load instruction came from).

4.2.2 Implementation of operations

The operation of this improved SSB is similar to the operation of the previous, simpler model described in Section 3.10, with operations classified and implemented similarly to Section 3.10.5, except for the addition of the forwarding logic. I highlight the differences below.

As before, the SSB intercepts operations, and handles them based on a real-time classification depending on whether they are loads or stores, and whether they come from a *finished*, *unfinished architectural* or *speculative* epochs, in the same way as shown in Figure 3.12. Previously, speculative granules could only be observed from the same epoch as the one that created them, whereas now younger epochs can also observe these values.

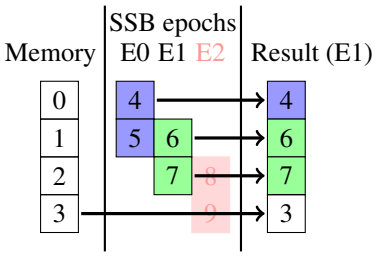
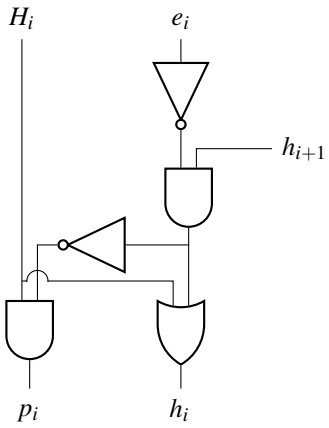


Figure 4.1: Lookup of speculative values. The load from epoch 1 (E1) observes the most recent value for each granule out of data in the memory system, E0 granules and E1 granules (where the memory system contains the oldest – architectural – values). Granules from younger epochs (such as E2) are ignored.



Inputs:
H_i: does threadlet i have a granule with matching address?
e_i: is i the threadlet issuing the read? (one-hot)
h_{i+1}: output 'h_i' of the next cell along a ring

Outputs:
p_i: will threadlet i respond? (one-hot)
h_i: will threadlet i *or a younger threadlet* respond?

Figure 4.2: Control circuit for the response of threadlet i to a read. There is one copy of this cell for each threadlet, linked together along a ring. Note that the logic is not circular, as h_{i+1} does not affect the output for the read's threadlet (i.e. if e_i is true).

Stores and any operations from finished epochs work exactly the same way as before, and loads from unfinished architectural or speculative epochs work very similarly, with slight modifications.

Speculative loads When the SSB lookup is performed (in parallel with the level-one cache lookup), we perform a lookup that returns the most recent version of each granule requested, from the requested epoch or any younger epochs, as described above and in Figure 4.1. This can be implemented by querying every granule cache slice for each access simultaneously, and using the simple combinational logic shown in Figure 4.2 (per threadlet context, connected along a ring in epoch order) to pick the most up-to-date version for each granule of the access. Once the architectural value is returned from the memory system, the SSB supplies any granules it has found a copy for (from the context i where p_i is set), and simply takes the rest of the values (i.e. if $\neg p_i$ for all i) from the memory system.

Loads from unfinished architectural epochs Handling loads from unfinished architectural epochs becomes easier under this improved implementation, since the cross-epoch forwarding method described above works for all loads, not just speculative loads. We perform an SSB lookup in parallel with the level-one cache read, looking for overlapping architectural granules from the SSB. Unlike for the simpler design before, immediate write-back of these overlapping values is not necessary, as the SSB lookup will correctly provide the appropriate (architectural) granules that are still in the SSB (waiting for write-back).

Slight care needs to be taken to avoid an edge case where a load never meets an architectural granule due to the timing of the write-back operation and the load. In particular, granules waiting for writeback are only marked, but not yet evicted when the writeback starts. Instead, they remain in the SSB until the writeback is confirmed via a response from the level-one cache. This way, there are three possible cases for the relative timing of a write-back and an overlapping load, as follows. If the load response comes before the writeback request, then the load reads the granules from the SSB. Otherwise, if the load request comes before the writeback request, then operation ordering in the level-one data cache (and memory system) ensures that the load response is sent to the SSB before the writeback response, thus the load will receive the values from the SSB granule correctly. Finally, if the load request comes after the writeback request, then the level-one data cache ensures correct ordering, and serves the load with the already-updated data.

4.2.3 Conflict detection

In order to take advantage of these improvements, we need to upgrade the conflict detector to only detect a conflict if a true read-after-write hazard occurs, and the sequential ordering is actually violated.

Logic

Suppose we have two operations targeting overlapping regions of memory, a read R and a write W (from epochs E_R and E_W, respectively). If R is executed after W, then no conflict may occur, due to the design of the forwarding logic. R either received the value written by W, or it was not supposed to observe W anyway (e.g. if W is younger in program order).

Otherwise, if $E_R < E_W$, then W is younger in program order, and the SSB correctly hides W from R. If $E_R = E_W$, then the LSQ and the memory system enforce correct ordering to comply with the consistency model. Otherwise, $E_W < E_R$. In this case, R should have observed W (and a conflict needs to be triggered), unless R received its value from a write W' that is younger than W in program order, i.e. $E_W < E_{W'} \leq E_R$. An example of this is shown in Listing 4.1.

Listing 4.1: The write W' hides write W from read R. R correctly observes the value V'.

```
1   W' : WRITE (epoch = 2, address = X, value = V')
2   R  : READ  (epoch = 3, address = X) → V'        // Correct! Ordering with W irrelevant.
3   W  : WRITE (epoch = 1, address = X, value = V)
```

All three operations target the same memory location, X. W' happens before R, thus R receives the correct value, V', using forwarding in the SSB. Then, W happens. Now, the correct return values for a read from X are as follows:

$$\textsc{Read}(E,X) \rightarrow \begin{cases} V & \text{if } E = 1 \\ V' & \text{if } E \geq 2, \end{cases} \tag{4.1}$$

following the forwarding logic shown in Figure 4.1. Thus, the return value of R is (still) correct, since W' *hides* W from R due to program order.

In case the order of the three operations is $R \rightarrow W' \rightarrow W$, then a conflict is correctly identified when W' is seen. If W' is executed last (i.e. the order is $R \rightarrow W \rightarrow W'$), then – semantically – a conflict should occur between R and W', as R should observe W' (and not W, as it is hidden). However, the conflict detector will already detect that a conflict is inevitable when W is checked. It cannot say whether W causes the conflict directly, or if there is still a future W' coming in that will cause the conflict, but this does not matter, as squashing is the correct response in either case.

Algorithm

Algorithm 4 shows the revised conflict checking implementation, accounting for value forwarding. The appropriate method is invoked whenever an operation is serviced by the SSB (in execution order).

For reads, we work out which granules were forwarded from the predecessor, and which were read locally from the same epoch. A granule is read locally, if it is already in the epoch's write set, since this means that there was a previous[1] write to it from the same epoch,

[1] 'Previous' in both program and execution order, since these are the same for such overlapping operations from the same epoch, due to the consistency model.

Algorithm 4 Eager conflict detection logic with eager forwarding

Require: Read and write sets $\textsc{Rd}(E)$, $\textsc{Wr}(E)$ for each epoch E

```
1:  function SpeculativeRead(Granules G, Epoch E)
2:      Fwd ← G \ Wr(E)    // Derive forwarded set.
3:      Rd(E) ← Rd(E) ∪ Fwd    // Granule read by epoch iff forwarded.

4:  function Write(Granules G, Epoch E)
5:      Wr(E) ← Wr(E) ∪ G    // Add to write set.
6:
7:      // Check for conflicts
8:      Fwd ← G
9:      for all e from E + 1 to YoungestEpoch()
10:         if Fwd ∩ Rd(e) ≠ ∅    // e observed stale value!
11:             Squash(e)    // Squash and restart e, recycle e + 1, e + 2, ...
12:             break
13:         Fwd ← Fwd \ Wr(e)    // All g ∈ Wr(e) are forwarded from e, not E.
```

which *hides* any writes from older epochs. Granules not in the write set were obtained from a previous epoch (i.e. forwarded) or read from the memory system, and therefore they are inputs to the epoch. Therefore, we add these granules to the read set.

As discussed above, conflicts can only occur if a read is executed first, followed by a write (from an earlier epoch), but not the other way around. Since the conflict detector processes operations in execution order, this implies that a conflict can only be discovered when processing a write, and so no conflict checking needs to happen on reads.

Upon executing a write, we first add the written granules to the epoch's write set. Then, the loop on line 9 goes through all younger epochs, and checks if any loads executed prior should have observed the newly-written value. It does this by iterating through epochs e, and keeping track of the set of granules *Fwd* that could be forwarded from this write to a read in epoch e. *Fwd* starts with all granules from the write. For each epoch e, we check if the forward set intersects with the epoch's read set. If so, a conflict is triggered, squashing e as well as all younger epochs (e is then restarted, while the successors are discarded and recycled). Finally, before moving on to the next epoch, line 13 removes all granules written in epoch e from the forward set. This step accounts for writes from e *hiding* the current write (from $E < e$) from any reads from epochs younger than e.

The update of *Fwd* is done after checking conflicts with e, as $\text{RD}(e)$ already accounts for writes from e hiding older writes (due to the set subtraction on line 2). This step is important, as it is the only way to keep track of ordering between reads and writes from the same epoch and thus distinguish between an epoch that first reads a value and then overwrites it (e.g. read-modify-write) and one that first writes to a location and then reads it back (e.g. a local variable).

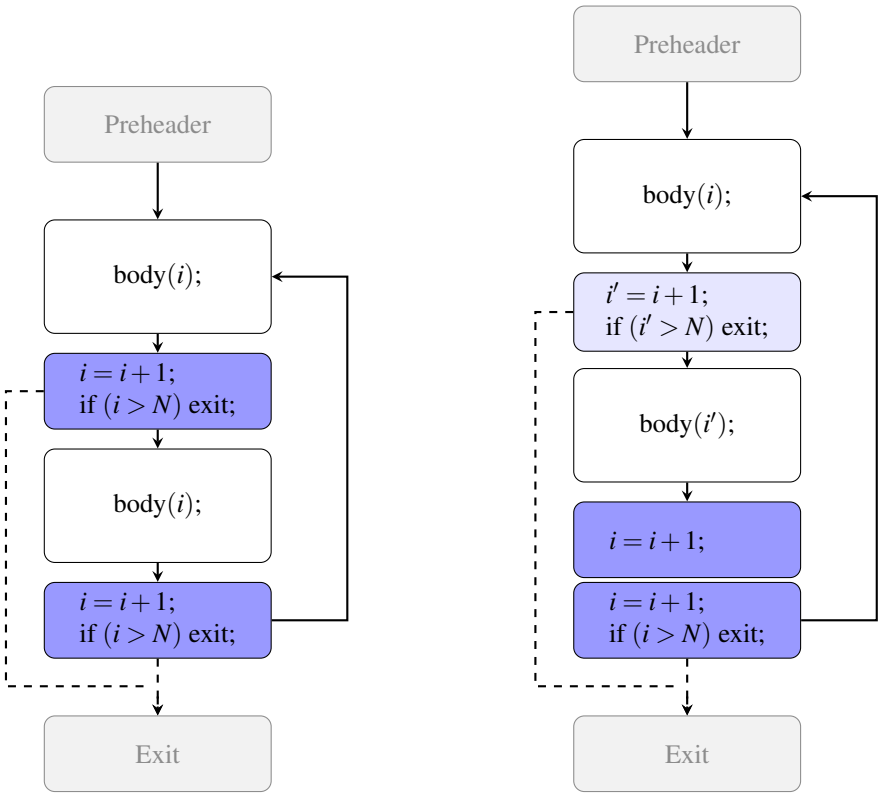
4.3 Iteration packing

Often, loops with largely independent iterations have a small loop body, containing only tens of instructions on average (sometimes with high variability). If iterations are too small, naïve parallelization will result in slowdown, as discussed in Section 1.4.1. In order to expose *additional* parallelism, speculative threadlets need to jump (far enough) ahead of the baseline CPU's out-of-order window to unlock opportunities for additional reordering. Therefore, increasing the size of epochs for such short loops is a potential solution for unlocking greater speedups.

4.3.1 Compiler-based solution

One way to achieve this goal would be to perform unrolling in the compiler. Although implementing such a pass, and solving the arising difficulties described in this section are beyond the scope of this thesis, I discuss the possibility here for completeness, and because it helps to present the idea of iteration packing through a known concept.

Being a well-known transformation, unrolling itself is easy to implement. In its basic form, unrolling does not directly expose larger parallel bodies, due to the placement of induction variable updates.



(a) Vanilla unrolling transformation

(b) Optimised for parallelism

Figure 4.3: Loop after compiler unrolling transformation. The base result on the left splits the parallelisable body into two parts, with a sequential induction variable update in the middle. The version on the right is optimised for parallelism by copying the induction variable update and localising the original copy to the current iteration (using the variable i', which is only live until the end of the second body). This way, both copies of the original body (and the new local induction variable update) can be merged into the new parallel body.

Figure 4.3(a) shows this phenomenon. After unrolling once, the (dark) induction variable update sections and the (white) parallel body sections alternate in the loop body. The hints given can only be inserted around one of the two body sections, as both induction variable updates need to be outside of the detached body region (that is, in the header or continuation). However, we can solve this by applying the code motion transformation shown in Figure 4.3(b). With this transformation, the three light-coloured blocks can all be part of the new body, and the two dark-coloured induction variable updates may be simplified and merged (e.g. to $i = i + 2$ in this case).

Unsolved issues and difficulties

There are a few difficulties in implementing and applying this method in practice. Firstly, due to restrictions on code motion. Secondly, due to the need to predict the dynamic size of the body, and thirdly, because of the possible impact on sequential performance.

Restrictions on code motion Problems may arise if the induction variable update contains memory accesses, or calls to functions that may contain memory accesses themselves. If memory accesses are (or may be) present, then the compiler needs to prove that none of the accessed locations can alias with any of the locations accessed inside the body, as moving the update past the body otherwise changes program semantics. Furthermore, we need to be able to duplicate the induction variable update safely, which may be difficult if the update has (memory) side-effects, and especially so if those side-effects occur inside a different function. These factors will – in practice – restrict the scope of this type of unrolling to loops with admissible induction variable updates. The complexity of admissible update patterns depends on the complexity of the compiler pass as well as fundamental limits.

Predicting dynamic instruction count The second difficulty is that the compiler needs to decide whether to unroll a loop or not, and if so, how many times it needs to unroll it to produce a body of appropriate size. To make this decision, two factors need to be calculated, which both present challenges in the compiler.

Firstly, the appropriate size needs to be defined, which will be dependent on low-level details of the microarchitecture (such as SSB size, latencies, pipeline depth and width, functional unit counts) as well as run-time behaviour of the loop (such as memory locality, speculative conflict frequency, instruction dependences) on the specific microarchitecture. Knowing such details in the compiler (middle end) is not a given, as the target microarchitecture is typically not known, and run-time behaviour depends on statically hard-to-compute or input-dependent factors.

Secondly, the dynamic size of loop iterations needs to be estimated. This is necessary in order to determine the necessary number of times to unroll in order to hit the appropriate size. Since the compiler has a static view of the program, it is unaware of run-time data, and it may not have visibility of all the code that may execute inside the iteration (via function calls to different compilation units), estimating dynamic instruction count is difficult and estimates are necessarily inaccurate.

Impact on sequential performance The third difficulty is limitations imposed by maintaining sequential performance. This matters for two reasons. Firstly, because serious degradation of sequential performance inevitably affects parallel performance too. Since we cannot expect more than a handful of threadlets (e.g. 4) to be feasible to add to a pipeline, and parallel speedups are limited by the under-utilisation factor of pipeline resources, parallel speedups are capped at a certain factor P ($\leq 4\times$). If the compiler needs to degrade sequential performance by a factor of S in order to unlock this parallel speedup, then the overall possible speedup will reduce to (around) $\frac{P}{S}$, which may be limited. If $P < S$, then parallelisation is unprofitable, and we would be better off not performing the enabling sequential transformations. Secondly, if the parallel speedup potential cannot be accurately estimated by the compiler, or if speedups are variable or unexpectedly low at run-time, then the loop is processed in sequential mode, and so slowdowns are endured. If the architecture supported code versioning, this impact could be somewhat limited, but it would not be reduced to zero due to the code footprint increase and additional branching (between versions).

Advantages of compiler-driven approach

With all of the above disadvantages, a successful compiler-driven approach could have the advantage of supporting complex induction variable updates. That is, the compiler can precisely identify what code is responsible for updating the induction variables, and it can correctly duplicate those instructions, even for an arbitrarily complex update pattern. In contrast, the microarchitectural approach described in the next section relies on prediction, thus it is fundamentally limited to simple-enough, predictable evolution. Furthermore, the microarchitectural mechanisms are relatively complicated, and a compiler-driven approach could simplify the microarchitecture significantly, leading to possible area and power savings.

4.3.2 Microarchitectural approach

Instead, we can perform a similar transformation in the microarchitecture, without relying on help from the compiler. Rather, the microarchitecture reinterprets the original architectural hints in a slightly different way, in order to perform more efficient parallelisation. Note that this is still a valid usage of the hints, as long as sequential semantics are respected, since the hints do not prescribe a single run-time usage. Instead, they provide additional information that the microarchitecture can take advantage of.

Idea

To increase the size of each epoch, the microarchitecture can choose to jump ahead on `detach` and *pack N* loop iterations into each epoch. When detaching from iteration I, the microarchitecture predicts the register starting state of iteration $I+N$. The successor is assigned to execute from $I+N$ onwards, and the current epoch will contain the iterations until that point.

This is similar to performing the unrolling transformation in Figure 4.3(a) dynamically, at run time, without modifying the code. The hints around the middle induction variable update section(s) are ignored (simulating their removal), and value prediction is used to eliminate the true dependency on the induction variable(s) between the body and the continuation.

Dynamic unrolling by out-of-order processors

At this point, let us briefly contrast iteration packing with 'dynamic unrolling', a behaviour exhibited by standard out-of-order processors. Since the out-of-order window may cover multiple iterations of a loop, out-of-order processors already – in effect – unroll loops dynamically (at run time), in order to find and expoloit parallelism between iterations.

For small and simple loops, we can expect this dynamic unrolling to already do a good job identifying and leveraging parallelism, as many iterations are present in the window simultaneously. If this is working well, then iteration packing clearly cannot help further, as pipeline resources are saturated, removing possible performance gains.

However, for (slightly larger) loops with hard-to-predict internal branching, we can expect the effective correct (not mispredicted) window size to be limited, and thus it would struggle to efficiently cover multiple iterations. If intra-iteration ILP is limited, iteration packing can enable threadlet-based speculation to expose additional independent streams of instructions

to saturate under-used pipeline resources. Furthermore, if high-latency memory operations are a bottleneck, then iteration packing can find more memory operations to execute in parallel, as well as useful arithmetic work to do while the pipeline would otherwise sit idle.

Implementation and predictors

We use the first few iterations of each loop to train up three predictors, which work together to control iteration packing. The first one estimates the size of each epoch to derive a good packing factor. The second predictor predicts the set of induction variables based on cumulative read and write sets, built up throughout the iterations. The third one predicts the values of these registers. Iteration packing is only performed if the value predictor can confidently predict all induction variable registers. The microarchitecture compares predictions to final values, and it safely updates mispredicted registers (if possible), or it squashes the successor otherwise. An example output from the predictors is shown in table 4.1, and the details of each predictor are expanded below.

Size predictor The size predictor logs the number of instructions committed by the current threadlet every time it reaches the reattach instruction.[2] If iteration packing is not being performed, then only one number is logged by the iteration. Otherwise, the counter is reset to zero, and it is logged and reset once per iteration in the epoch.

These logged numbers form a stream of iteration sizes, which can be used to predict the size of future iterations. Predictions need not be constant, or even particularly smooth, as the packing factor can easily be varied on an epoch-by-epoch basis, by storing one counter per threadlet context. Instead, it is beneficial if the predictions are able to respond to changes in the behaviour of the loop. Programs often contain loops whose iterations become generally larger or smaller over time. Examples of such loops include convergence algorithms, graph traversals, sorting and other polynomial-time array operations.

[2]The very first iteration will be slightly smaller than the rest, as it starts at the loop entry and not the continuation. This iteration can be ignored for training or used as is, and this has no noticeable impact on overall performance.

Size predictor	≈ 234 instructions per iteration
Induction variable predictor	r10, r12
Value predictor	$\text{r10} = 4 \times i + 465282$, *confident*
(iteration i)	$\text{r12} = -1 \times i + 1024$, *confident*

Table 4.1: Example output from the three predictors. In this case, iteration packing can be performed, as we have confident predictions for all induction variable registers (r10 and r12). The packing factor is chosen based on the desired epoch size and the predicted iteration size. For example, if the target is 'at least 1000 instructions per epoch', then we may choose the packing factor of 5, as $5 \times 234 > 1000$. When detaching from iteration 23, we'd then jump ahead and start iteration 28, with the starting state of $\text{r10} = 4 \times 28 + 465282$ and $\text{r12} = -28 + 1024$, and all other registers kept the same as inherited from the detach point.

A simple method for responding to changes and deriving a current estimate is by calculating a running average, with exponential decay. For example, if s_i is the size of iteration i, then we can derive S_{i+1}, the estimate for the size of iteration $i+1$ as

$$\begin{aligned} S_{i+1} &= \alpha \cdot S_i + (1-\alpha) \cdot s_i \qquad \textit{for } i \geq 0 \\ S_0 &= s_0 \end{aligned} \tag{4.2}$$

for a decay factor α. The above expands to

$$S_{n+1} = \alpha^n s_0 + (1-\alpha) \sum_{i=1}^{n} \alpha^{n-i} s_i \tag{4.3}$$

which has three nice properties. Firstly, it is easy to implement the update logic, using two multipliers and an adder. Secondly, if the loop iterations are similar, say if s_i is constant for all i, then the result stays constant at $S_i = s_i$. Thirdly, if the behaviour changes, say $s_n = x$ for $n < N$ and $s_n = y$ for $n \geq N$, then for $n > N$, S_n quickly converges to y, the new iteration size. That is because the factor α^{n-i} in eq. (4.3) converges exponentially to 0 as $n-i$ grows. Thus, the contribution of terms $s_0, \ldots, s_N$ reduces exponentially for later iterations n. Similarly, the chosen default value of S_0 is not important, as its impact diminishes to zero exponentially.

In summary, this scheme handles well any loops with stable, increasing or decreasing iteration sizes, as well as those with phase behaviours. Two common loop iteration size distributions I observed in benchmarks that are not handled well are bimodal (e.g. loops containing an if-then-else, with either the 'then' or the 'else' path being significantly shorter), and long tail distributions (e.g. loops with an inner loop whose iteration count is drawn from a long tail distribution). These could be handled using other schemes for calculating the rolling average (that model cases better). Switching between predictors based on performance on the current loop is also possible, but this space was not further explored in this thesis.

The result of the size predictor is used to dynamically set a packing factor. The packing factor P is chosen as the smallest value resulting in reaching a target threshold T of dynamic instructions per iteration (for example 1024) based on the prediction. That is, we set

$$P_i \triangleq \left\lceil \frac{T}{S_i} \right\rceil . \tag{4.4}$$

Since this depends on S_i, it will change over the lifetime of the loop. The microarchitectural iteration packing scheme can handle this, as it can handle an individual packing factor per epoch, and the dynamic calculation gives us the ability to respond to changes in the run-time characteristics of the loop.

Induction variable predictor This predictor approximates the set of induction variables based on register read and write sets. This is important in order to limit the scope of value prediction. Apart from wasting energy and chip area[3], attempting to predict the value of irrelevant temporary registers may lead to low value predictor confidence, and thus prevent the microarchitecture from attempting iteration packing, or it causes squashing due to incorrect predictions.

[3] In a real implementation, we would likely only add a handful of value predictors.

Let us define the induction variable *update section* U_i as the dynamic region of the instruction stream containing a continuation and the successive header (but excluding the parallel body B_i). Let us record the body and update section read and write sets (R_B, W_B, R_U and W_U, respectively) by recording the read and write sets for each section, and taking the union over all bodies and all update sections. That is,

$$R_B \triangleq \bigcup_i R_{B_i} \qquad W_B \triangleq \bigcup_i W_{B_i} \tag{4.5}$$

$$R_U \triangleq \bigcup_i R_{U_i} \qquad W_U \triangleq \bigcup_i W_{U_i}. \tag{4.6}$$

As before (see Section 3.11), the read set should refer to input values being observed, thus a register t that is first (fully) overwritten and then read in a section S should be in the write set but not the read set (i.e. $t \in W_S \wedge t \notin R_S$).

Let us classify registers as either constants, temporaries and outputs or induction variables, based on the write and read sets in the update sections and the parallel bodies over the lifetime of the loop.

Constants c are not written to in the loop, that is,

$$c \notin W_B \cup W_U. \tag{4.7}$$

Temporaries and outputs t are not in the read sets, but they are written to. That is,

$$t \notin R_U \cup R_B \quad \wedge \quad t \in W_U \cup W_B. \tag{4.8}$$

Note that the first condition here means the register's input into the sections is never observed. Values written in the same section can be read back from the register. The distinction between outputs and temporaries does not matter for training the predictor, but make the naming more sensible. (Temporaries are initialised then read within individual sections, whereas outputs are read after the end of the region, after the *sync* instruction.)

Induction variables i are all remaining registers. Using the definitions above, we get

$$i \in W_U \cup W_B \quad \wedge \quad i \in R_U \cup R_B. \tag{4.9}$$

However, register dataflow restrictions forbid values created in the body being read in a later section (until the end of the region), thus[4] $i \in W_B \implies i \notin R_U \cup R_B$, and so we have

$$i \in W_U \quad \wedge \quad i \in R_U \cup R_B, \tag{4.10}$$

or equivalently,

$$i \in W_U \cap (R_U \cup R_B), \tag{4.11}$$

We notice that only induction variables (defined as above) need to be predicted for correct execution, as correctness for the other two types is ensured by other mechanisms. In

[4]This ignores the edge case where i may be read and written in the body, but the reads are in iterations before the writes. That is, $\exists j.(\forall k \leq j.i \notin W_{B_k}) \wedge (\forall k \geq j.i \notin R_{B_k})$. The microarchitecture will mispredict if this happens, but this is acceptable, as it is not a natural pattern. In particular, the compiler prototype will never generate it.

particular, for constants, the value is the same between the detach point and the start of the continuation in any iteration, for temporaries and outputs, the value of the register is not an input to the next epochs, as the read sets of those epochs do not contain the register. Since register value merging (see Section 3.2.4) ensures that the value read after the end of the region is correct, the stale starting values do not affect execution, and they can be ignored.

The induction value predictor simply tracks the register read and write sets for each body and update section, and merges these with the global body and update section read and write sets once a new iteration's relevant section completes. This gives an estimate of the read and write sets across the lifetime of the loop. Until all possible control paths in the loop are seen, these sets may be incomplete. This is not a problem, since we only use them for prediction and in practice the first few iterations produce a good approximation. Note that the prediction does not need to be safe, as correctness is checked before each epoch commits, and thus squashing can be initiated in case of incorrect predictions.

The very first iteration of the loop is handled specially, as its entry point (and hence read set) is different from other iterations, which could lead to the unnecessary inclusion of additional registers in the update section's read set.

Value predictor The value predictor is trained and queried for each induction variable register identified by the induction variable predictor. Although any known value prediction technique can be used here, the most important pattern to recognise is a simple stride. That is, the value of induction variables often depends linearly on the iteration number. This is common due to patterns of incrementing a loop counter, or iterating through an array.

The prototype implements a simple stride predictor with a confidence counter. Whenever a new iteration is committed, the starting value is saved as the 'last starting value'. If the starting value matches the prediction, then the confidence counter is incremented. Otherwise, the starting value and stride are calculated using the current start value and the last logged start value, and the confidence counter is reset to zero. The prediction is considered confident if the counter has crossed a given threshold. This reset and re-train mechanism is designed to recover after 'blips' in the stream of induction variable values, such as infrequent conditional updates (e.g. iterate through array skipping a few elements), and it is a common technique borrowed from branch predictors.

In order to perform iteration packing, each of the value predictors has to give a confident prediction, so that the input state can be derived with high confidence for each epoch. The prototype implements one predictor per general-purpose integer architectural register, but a real implementation might opt to only have a handful of predictors that can be assigned to the registers that are predicted to hold induction variables (unless there are more induction variables than predictors, in which case prediction is not possible). Floating point and vector registers are not predicted as they are unlikely to contain induction variables, and their prediction would be more challenging (due to complexity and size, respectively).

Out-of-order training

We can avoid buffering state for a long time by relaxing the training models.

The induction variable predictor does not mind the order of receiving training inputs, as it takes the unions of all read sets and all write sets it receives. The size predictor can be trained on slightly out-of-order inputs as well. Assuming inputs are only reordered within a window of a few of iterations (e.g. those overlapping in time), the predictor presented above will still produce approximately correct results at all times, and it will correctly capture constant and phase behaviours (with a slight delay in convergence after a phase shift).

For the value predictor, we can also enable out-of-order training at the cost of some additional approximate behaviour. When the predictor is trained, and values logged match the predictions, we can just increase confidence and continue as usual. If a logged value disagrees, we can reset confidence, save the current logged value and the next one, and use these to re-calculate the starting point and stride. Then, any future predictions can be validated against predictions obtained from this new trained state, until the confidence counter has hit a high enough threshold. It may be that some iterations disagree between the training set and the latest iteration, but this scheme still works well for a constant stride and phase behaviour (and the case when the stride is mostly constant but occasionally 'slips' due to a conditional update).

Analysis of the microarchitectural approach

This approach is able to closely track run-time characteristics of loops, and dynamically adjust the packing factor to keep iterations big enough, but prevent them becoming too big. An overly long iteration could overflow the speculative state buffer, and lead to additional conflicts. Unlike the compiler-driven solution, it also does not result in any changes to the machine code (which may lead to sequential slowdowns), or any static analysis (which may limit the scope). On the other hand, induction variables must follow a strided pattern, thus loops with non-linear patterns (such as linked list traversals, or loops that increment the induction variable conditionally or in an inner loop) are not supported with the current stride predictors. Although more complex predictors are possible, they would need to be designed for a specific domain of loops, and there will always be loops that are not covered by our predictors.

4.3.3 Growth in complexity

It is worth noting that performing iteration packing increases complexity, whether it is done in the compiler or in the microarchitecure. As discussed earlier, the compiler needs to predict performance and iteration size in order to perform unrolling profitably, which may require profiling and lead to issues about portability and sequential code quality. Alternatively, the microarchitecture needs to track a large amount of state, including register values, and perform (simple) value prediction in order to orchestrate iteration packing. As we will see in Section 4.4, this can yield a significant additional uplift in performance, but there is definitely a trade-off between performance and complexity. Deriving the optimal choice between applying none, one, both or a combination of the two approaches for iteration packing requires far more involved engineering and analysis than presented here, and it is left to future work.

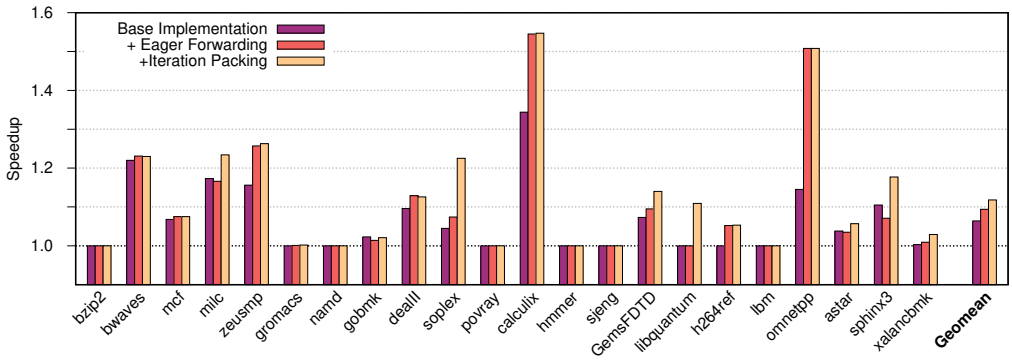


Figure 4.4: Effects of the two optimisations. The baseline scheme achieves a geometric mean speedup of 6.4%, eager cross-epoch value forwarding improves this to 9.4%, and iteration packing further improves the speedup to 11.8% across the SPEC CPU2006 benchmark suite.

4.4 Results

Figure 4.4 shows the improvements achieved by the two optimisations presented in this chapter. As we can see, the geometric mean speedup improves with each optimisation, as do the individual speedups of many benchmarks. Some benchmarks show small regressions on individual optimisations. As discussed below, this is due to repeated squashing for early forwarding, and increased conflict rates for iteration packing. The only significant (more than 1%) regression above is a 3% drop in performance for *sphinx3* for eager forwarding, and it is due to approximation error and numerical instability in the loop selection process, but the losses are re-gained in the final version.[5]

Regressions are observed on some individual loops, and small regressions are shown on a handful of benchmarks. Regressions on cross-epoch forwarding occur because this optimisation may lead to heavily-conflicting epochs restarting many times, thus not making meaningful progress. This can limit the positive prefetching effect of incorrect speculation. Iteration packing may increase conflict rates, since any one of the packed iterations may conflict, leading to a squash. These losses can likely be removed by good engineering and fine-tuning of parameters in production, but this is beyond the scope of this thesis.

4.4.1 Source of gains

The improvements shown above are achieved by increasing the number of profitable regions, and by increasing performance on a handful (3 and 4, respectively) of already-profitable, high-coverage loops.[6] To generate the plots in this section, I re-ran loop selection (as described in Section 3.12.1) for each configuration. Figure 4.5 shows the changes in the

[5]Two loops are active in most of the same loop-selection buckets, and thus they show up as overlapping to the loop selector, even though they are in fact disjoint. A 0.1% difference in speedup predictions for one of the loops leads to loop selection picking both loops in the base and iteration-packing versions, but only picking one when eager forwarding is enabled but iteration packing is not. Since the gains are disjoint here, picking both is better.

[6]Counting over 1% whole-workload impact from each optimisation on loops that already brought at least 1% benefit without the given optimisation. The 7 loops each cover over 50% of their respective workloads, bringing $5-15\%$ additional whole-workload benefit each.

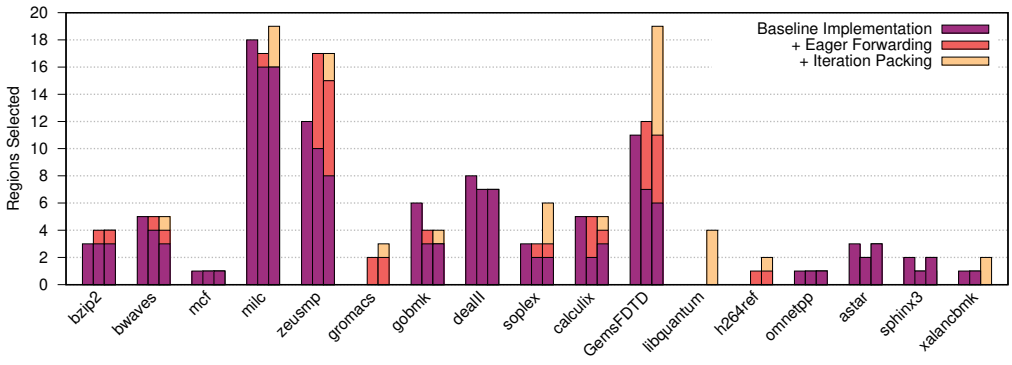


Figure 4.5: Selected profitable regions per benchmark in the three versions (benchmarks with no regions selected are hidden). The three bars correspond to the three different versions. Bars are coloured based on which version covered a region first. For example, in zeusmp, we selected 12 regions in the base case, out of which 10 were still selected with eager forwarding and 8 with iteration packing, 7 new regions were also selected with eager forwarding, out of which all 7 were still selected with iteration packing, alongside 2 new loops that were only chosen with iteration packing.

regions picked per workload. Compared to the 79 profitable (static) regions picked for the base configuration (as most optimal), eager cross-epoch value forwarding incorporated 23 new loops, but did not pick 19 of the original 79, thus picking 83 loops in total. The final version, with iteration packing enabled as well, added 25 new loops that were not picked in either of the previous versions; it picked 58 of the 79 loops from the base version, 21 of the 23 new loops that were enabled by eager value forwarding, giving a total of 104 loops chosen for parallelisation. Loops may be de-selected following the addition of new optimisations because their performance regresses due to the optimisation (as is the case for *calculix*, *astar* and *sphinx3*), but more commonly, this is due to another overlapping loop yielding better speedup potential. For example, eager forwarding enables some outer loops to be chosen (by increasing their performance), and iteration packing enables some inner loops to get good gains where these inner loops were unprofitable and the outer loop was barely profitable before.

4.4.2 Prediction accuracy for iteration packing

Out of the 259 unique (static) loops tested during loop selection, 171 had fewer than 1000 instructions per average iteration in any one SimPoint region. Out of these, 129 were successfully unrolled with the remaining 42 failing due to a lack of predictor confidence (due to non-linear loop-carried dependences or the regions being too short to gain confidence). Out of the successful ones, only 10 experienced unrolling mispredictions (where the predictor was confident, but gave predictions that later turned out to be incorrect in some iterations), and for 8 of these, the number of mispredictions was likely negligible (i.e. fewer than 0.2 mispredictions per million instructions).

4.5 Other optimisations

Here, I describe a range of other optimisations attempted in order to increase performance. These either did not have a significant impact on performance, or their impact was already covered by the two main optimisations. In other words, adding the optimisations to the initial design from Chapter 3 may yield some uplift, but applying them to the design evaluated in Section 4.4 does not.

4.5.1 Partitioning of resources

The scheme discussed so far prioritises the older threadlets over the younger ones. When allocating bandwidth (e.g. in the fetch, issue or commit stages), this is straightforward: we try to process the oldest instruction that is ready or eligible to be processed across all threadlets. Since the threadlets are strictly ordered by epoch number, if an instruction from an older threadlet is ready, then it should be prioritised over instructions from younger threadlets.

However, this becomes more complicated for space-based resources (i.e. buffer space, queue entries or physical registers). The issue is that allocations can last for many cycles, and allocating more entries may not increase performance at all. For example, if there are 10 entries left in the ROB, and the next 10 instructions will each have to wait 200 cycles or more for their source operands to be ready, then there is no use using up 10 ROB slots now to allocate these instructions. Instead, adding 10 instructions from other threadlets to the ROB may lead to them being able to issue immediately, thus making progress.

One could argue that – since we expect the scheme to work well on predominantly low-ILP regions – it is worthwhile to strictly limit the allocation of each threadlet in the back-end buffers. While older threadlets can still be prioritised for bandwidth allocation, eventually, once they have filled back-end buffers with instructions, back pressure will naturally allow younger threadlets to take their share of bandwidth. Since the older threadlets have their share of back-end buffers full, most of the inherent instruction-level parallelism is already exposed, thus blocking the front-end (due to a full buffer) should not have a significant impact on performance.

This prioritisation scheme is discussed in more detail in Section 5.2.6, in the context of avoiding deadlocks due to new blocking operations introduced in Section 5.2. Without those instructions, priority inversion cannot lead to deadlock, as the progress of each threadlet in each pipeline stage can only be blocked by back pressure from the next stages, or if the threadlet has fully drained (see Section 3.6) after reaching the final reattach or sync. Thus, the only motivation for partitioning here is to increase performance.

However, upon implementing the prioritisation scheme, we find no significant overall changes in performance, only very small variations in individual benchmark performance. The reason is likely that limiting space can limit within-threadlet instruction-level parallelism, and partitioning allocations will lead to wasted space due to fragmentation. Furthermore, I found that an asymmetric partitioning policy (i.e. statically allocating more entries to the architectural threadlet) always lead to a reduction in performance.

It is possible that a more advanced partitioning scheme could work better. For example, the system could start enforcing the partition limits only if the buffers risk getting full. Since the design space is quite large, this is left to future work.

4.5.2 Early detaching

In order to be able to exploit fine-grain parallelism, the overheads when starting new threadlets need to be minimal. For highly out-of-order processors, it can take *detach* instructions many cycles to propagate through the whole pipeline, especially in a low-IPC speculative thread. Therefore, it is important to execute detaches as early as possible in the pipeline. The scheme presented in this thesis only detaches new threadlets once the *detach* instruction has made it to the front of the ROB (i.e. it detaches *in commit*).

Waiting this late has the advantage that all the input registers will have been produced and written back to the register file, thus we can copy the commit rename map and take a checkpoint of values in case the new threadlet needs to be squashed. Let us consider bringing detach forward to register rename. Even though we will not have the final values, we have performed register rename on all prior operations already. Thus, we know *where* all the input values will be produced to. For example, R_n (architectural register n) at detach time may be produced by an instruction I just prior to the detach, but when the detach is renamed, we will know both that R_n will map to P_m (physical register m), and from the scoreboard we can tell that P_m has not been produced yet.

We can certainly use this information to start fetching, decoding and renaming instructions from the (new) successor threadlet, as these operations do not depend on the values stored in the registers. Furthermore, if we share a scoreboard across threadlets, then we can even issue, execute and write back any instructions for which the sources are ready. Note that in this version, physical registers can be shared by multiple threadlets, but note that the register still refers to a single logical location and it has a single producer. We can tweak the register reclamation logic in order to correctly free and recycle physical registers only when no users remain. Additionally, there are two edge cases around speculation and squashing that need special attention.

Firstly, previously branch mispredicts inside any of the threadlets only ever led to squashing instructions within that threadlet, and they did not affect successors. However, if a detach instruction has been taken in register rename and it is subsequently squashed, then the successor(s) need to all be squashed too, as they were launched in error. This situation can be handled by the standard threadlet squashing logic once identified. For identification, we can add a tag to the detach instruction to say it has been speculatively taken, which can be noticed when instructions are squashed following a branch mispredict. Alternatively, a single register can store the sequence number of the last speculatively taken detach within each threadlet, which can be compared to the offending branch's sequence number and the last committed sequence number.

Secondly, since not all register values are ready at detach time, we cannot take a full register checkpoint, containing values straight away. However, we can checkpoint the rename map instead. Alternatively, we can either delay committing instructions in the new threadlet until the checkpoint has been taken, thus ensuring that no conflicting reads can be committed

(thus squash and restart are not needed), or simply throw away the threadlet in case it needs to be squashed.[7].

In the prototype, I implemented a version based on register aliasing and delayed instruction commit in case of speculative detaches (as described above). This lead to a significant reduction in slowdown for many unprofitable regions, and some minor speedups for profitable ones. However, iteration packing covered all of those gains. This is not surprising, as the two techniques both target loops with short iterations.

4.5.3 Short body skipping

Another class of regions where parallelisation suffers is those where the size of parallel bodies are not constant. For example, the bulk of the work may be inside a conditional statement or inner loop. In case the work is skipped in certain iterations, then the body may only consist of a handful of instructions, whereas in other iterations it may have thousands.

Notice that detaching during the short iterations is likely detrimental to performance due to the overheads of launching and retiring the new threadlet. This is especially true if the outcome of the skipping branch is correctly determined in the front end. In particular, by the time the detach would be taken (and instructions from the new threadlet fetched) the old threadlet has already reached and passed the reattach point in the instruction fetch phase.

If the above case occurs, we can detect it easily and elide the detach. If the prediction was correct, then this leads to a net gain, and allows the additional threadlet context to be used for a bigger future iteration, where the work in the body is not skipped. Adding this detection to the pipeline is straightforward. The instruction fetch stage can simply compare the program counter values fetched from to the ID of the current active region (which is the address of the continuation), and store the last cycle or instruction sequence number when this address was passed. This can then be compared to the fetch time or sequence number of the detach instruction to figure out if the start of the subsequent (corresponding) continuation has been fetched.

Short body skipping and early detaching together increased speedups for the libquantum benchmark by an additional 7.5 percentage points (to 18.4% total), however, these optimisations yielded a 20% reduction in speeedup for calculix, bringing the overall speedup number down. Additionally, both optimisations significantly reduced slowdowns for many unprofitable regions, however these still remained unprofitable or overlapped by a more profitable parent loop, thus yielding no additional improvements.

4.6 Summary

I presented a two important optimisations, which improved the overall performance of the system, increasing speedups from 6.4% geometric mean to 11.8%, over the SPEC CPU2006 benchmark suite. I also described a range of other optimisations that did not make a significant impact after the first two optimisations were applied.

[7] In the last case, the predecessor cannot be recycled as it needs to continue executing the program if its successors are thrown away.

Chapter 5

Cross-epoch dataflow

Chapters 3 and 4 presented an architecture and optimised microarchitecture to perform TLS and gain good performance for certain loops.

However, there are two types of loops that are not performing well. Firstly, there are loops that are successfully parallelised by the compiler, but cross-iteration memory conflicts prevent us from realising gains. Secondly, there are loops where the compiler struggles to transform the loop due to cross-iteration register dependences between the body and a future iteration.

We have to recognise that some cross-iteration dependences are fundamental, and there is no silver bullet for addressing them all. In particular, if the critical path is through a dependent chain (with no special properties), then it is impossible to gain performance without algorithmic improvements. Instead, this chapter investigates some ways that specific types of conflicts can be eliminated, reduced or mitigated. I propose some architectural and microarchitectural solutions, but – notably – I leave the necessary compiler support to future work. Thus, to evaluate the microarchitectural and architectural components, I test them on hand-crafted working examples, instead of running full regressions.

Later, I also consider other types of dependences, and suggest ways that future work may be able to handle them, in order to realise further gains, and parallelise more loops.

5.1 Associative updates (reductions)

One common special type of conflicts in loops is reduction operations. During a reduction, an output value is computed by combining some input values using an associative (and often commutative) operator (e.g. summing an array), such that intermediate values are only used for updating the total (and not for doing other dependent computations within each iteration).

However, due to the associative properties of the reduction operator, reduction-based hazards can be resolved without enforcing a strict ordering. If our system can effectively re-bracket such expressions, then conflict checking can be relaxed, and squash rates can be reduced.

Although reductions are typically defined to refer to a specific pattern of using a single associative update operation to derive a single output value at the end of the computation, in this section I use the word *reduction* more broadly, to refer to locations updated using

associative update operations in a loop, without being read in any other way in most iterations. These patterns are a natural extension of the reduction paradigm and appear similar at run time, therefore considering them leads to easy gains in performance.

5.1.1 Motivation

I investigated the instructions triggering conflicts at runtime in the GAP benchmark suite [4]. The code for these benchmarks is far easier to analyse and change by hand, making them ideal candidates for deep dives, but the workloads are still large enough to be interesting. I observed reduction-style conflicts for 5 of the 18 promising loops identified by hand. Loops with reductions were present in 3 different benchmarks (*bc*, *bfs* and *tc*). Examples include summing values over the neighbours of a node in the graph, and counting the number of nodes matching criteria.

A good example is the *tc* (triangle count) benchmark, which cleverly iterates[1] over 3-tuples of nodes that may form a triangle (i.e. a set of 3 nodes where each pair is connected by an edge), using three nested loops. The only loop-carried dependence in either of the two outer loops comes from the variable named *total*, which counts the total number of triangles found so far. This variable is incremented in a conditional statement inside the innermost loop.

5.1.2 Update operations

In theory, any commutative operations targeting memory locations or registers can be detected as reductions. In practice, some operations are easier to handle than others.

The most common examples in benchmarks examined (in GAP and SPEC CPU 2006) are integer addition/subtraction, bitwise logical operations (e.g. and, or, xor), maximum/minimum calculations, and floating-point arithmetic operations (e.g. addition/subtraction, multiplication).

Integer addition and subtraction (which are logically the same operation) and bitwise logical operators are simpler, as the update itself is encoded as a single instruction that is easy to pattern-match either in the compiler or the microarchitecture. For example, for bitwise-or, the AArch64 assembly operation is `or x1, x1, v2` or `or x1, v2, x1`, where `x1` is the register (currently) holding the reduction variable (i.e. partial sum), and `v2` is any value (either immediate or register), not derived from `x1`. In a compiler's intermediate representation, the update shows up similarly.

For minimum and maximum, there are two possible implementations: the code either uses a conditional select instruction, or it uses branching. Conditional select behaves similarly to other reductive operations, except the matched pattern contains two different instructions. The first instruction sets the condition, while the second one uses the condition to select the right value. The branching-based implementation, on the other hand, is significantly harder to support,[2] as convergence of code paths and the lack of other control-dependent instructions

[1] Starting with an edge (u, v), the code simultaneously iterates over the neighbours of u and v to find common neighbours, exploiting ordering to reduce complexity to $O(VE)$ for V nodes and E edges.

[2] This was not fully explored in the prototype, with the assumption that the compiler could make sure not to output such code in loops of interest if needed.

needs to be tracked or proven (by the compiler, microarchitecture, or a combination of the two).

Floating-point operations present the issue that they are not fully associative, due to rounding [23]. Since the floating-point representation is unable to support infinite precision, intermediate results may need to be rounded to the nearest representable value. This leads to possibly different results, and a loss of numerical stability, if the order of operations is changed. In this way, we cannot treat floating-point accumulation as a reduction by default, as doing so will break certain algorithms. However, such optimisations are permitted, and routinely performed in many cases using program annotations or compiler flags (such as llvm/clang's `__FAST_MATH__` macro and `-ffast-math` flag [82]). In particular, we managed to successfully compile and run SPEC CPU2006 benchmark suite in full with `clang-16` with the `-ffast-math` flag enabled (`-O3` optimisation, AArch64 target). Adding this flag did not introduce any new validation test failures.

Thus, future work can explore annotating performance-critical reduction variables in a similar way, then perform reduction-handling transformations in the compiler using these annotations, or pass the information down to the microarchitecture, so that the reduction is handled at run-time just like integer reductions are.

The prototype only demonstrates a working system for integer reductions, and it leaves minimum/maximum calculations and floating-point reductions to future work due to these complexities.

5.1.3 Scalar reductions and array reductions

We can classify reduction patterns into two types. The simpler type is scalar reductions. Here, a value – either held in a register or in a single variable in memory – is updated using a reduction operation. This type is the one more often considered as standard, due to its simplicity (e.g. OpenMP pragma annotations allow the tagging of scalar reductions). The second type is array reductions. A common example [22] is a histogram-style reduction, such as the very simple one shown in Listing 5.1.

This is an array-based reduction with the reduction operator + and the added value being the constant 1. Each element of the array evolves individually, following a reduction pattern. More complex examples also exist in benchmarks, with multiple updates, and varying increment values. For example, array values are often incremented based on the neighbourhood of the current node in the GAP benchmark suite.

Listing 5.1: Simple histogram reduction. One element of `A` is incremented each iteration, based on the value of `B[i]`, which is not known at compile time.

```
1    for (int i = 0; i < N-1; i++) {
2        A[B[i]]++;
3    }
```

5.1.4 Detecting reduction patterns at compile time

Present state-of-the-art production compilers (such as LLVM) have a very strict definition of reduction patterns. Extending the set of recognised patterns has been explored in past work [22, 31]. As always, challenges arise from the requirement for the compiler to prove independence before safely performing code transformations. For example, if a reduction variable is stored in memory, then – without further analysis – any memory accesses could overwrite or observe the reduction variable, breaking the pure reduction pattern. In the case of traditional multithreading or vectorisation, this would lead to incorrect execution, thus the compiler either needs to inject runtime checks, or avoid unsafe transformations altogether.

With threadlet-based speculation, these requirements could be relaxed. Suspected or confirmed reductions could be annotated by the compiler, and then reduction patterns could be handled by some combination of explicit compiler-inserted synchronisation and microarchitectural tracking and detection.

One promising approach is described by Han et al. [31], who aim to identify and exploit what they call *partial* reduction variables (PRV). These are variables that are updated like reductions in at least one place, but may (or even must) alias with other accesses. They then perform speculative parallelisation based on privatisation, and sequentialised updates at the end of each task (after the task became *safe*, which is equivalent to our definition of becoming the *architectural* threadlet). Extra hardware is used to track non-reduction-style loads and stores affecting a reduction variable within the same task, and these accesses also lead to synchronisation. This approach relies on avoiding conflicts using clever synchronisation stalls and private copies of variables, however – as the authors acknowledge – the synchronisation cannot handle all cases. Furthermore, the cost of privatisation, copying and synchronisation would likely be higher in our case, due to longer epochs (which have larger data footprints), necessary when targeting modern, and state-of-the-art large out-of-order CPUs (for the reasons discussed generally in Chapter 2). Thus, this approach – and similar approaches – could work well together with microarchitectural techniques, or a co-design based on compiler-based pattern detection and microarchitectural handling.

Detecting in-register reductions is an easier task compared to detecting in-memory reductions. On the contrary, updating values and checking for conflicts in the microarchitecture is easier for the in-memory case, as we can use the SSB. Furthermore, handling in-flight operations that observed a location is also easier in this case, as described below. The next subsections describe how the prototype detects, handles and conflict-checks in-memory reductions.

5.1.5 Update sequence for in-memory reductions

Since memory locations cannot directly be targeted by arithmetic and logical instructions, detection here tracks load-update-store-kill sequences, as these can be used to update reductions in memory. The load copies the value from the reduction variable in memory to a register, the update applies a reduction operation to that register, and the store copies the value back to the same address. Furthermore, we require that none of the intermediate values are observed by any other instructions. The kill instruction then kills (overwrites) the value in the loaded register, ensuring that it cannot be observed by instructions later in the program.

Without a kill instruction, it would still be possible to track reductions, but the hardware requirement would increase in order to correctly keep track of edge cases, such as spilling a dead value from the register to the stack.[3] If compiler annotations are added, this requirement could be elided, accepting the undefined state of spill slots, or explicitly wiping registers on the store. While a suitable kill instruction often exists in the unmodified binary (when the dead temporary register that was used for the reduction is re-used), it is usually too far away in the dynamic instruction stream to be usable as is. Instead, minimal compiler or manual modification is required; we add a kill instruction at the end of each simple load-update-store sequence in annotated loops. In the current prototype, this is done manually, but a compiler back-end pass could easily do this.

Listing 5.2 shows an example of a recognised reduction update sequence. The comments summarise the restrictions imposed by the prototype microarchitecture implementation. We require that the load and store operate on a single memory location and register, where the memory location is encoded without pre- or post-increment notation, as a register-indirect address.[4] The address encoding has to match exactly between the load and the store instruction, and the referenced register cannot be written to between these two instructions (i.e. the same physical register is used in both instructions). The load and store also have to operate on the same architectural register for the reduction value. The kill instruction needs to fully overwrite the register (e.g. if the reduction value is 64 bits long, then kill needs to write a value to the whole register). The update instruction has to be in the format shown in Section 5.1.2 (e.g. `or x1, x1, x3`).

[3] This would leave a copy of the reduction from midway through the epoch in memory. To preserve sequential semantics, this memory value needs to be recalculated and updated if the starting (input) value of the reduction changes.

[4] Other addressing modes can be supported trivially, but this covers the common case for AArch64.

Listing 5.2: Example in-memory reduction update sequence.

```
1   // Load: single register-indirect location into register.
2   ldr x1, [x2, #100]
3         ...
4         // No access to x1. No write to x2. No conditional/indirect branches.
5         ...
6   // Update: recognised opcode, target and one source matches x1.
7   add x1, x1, x3
8         ...
9         // No access to x1. No write to x2. No conditional/indirect branches.
10        ...
11  // Store: register and address (encoding) matches load exactly.
12  str x1, [x2, #100]
13        ...
14        // No read from x1. No conditional/indirect branch.
15        ...
16  // Kill: fully overwrites x1, source registers (x5, x6) are independent of x1.
17  add x1, x5, x6
```

Lastly, no conditional or indirect branches, and no barriers of any type, may exist between the load and kill, as these could result in complicated squash and replay patterns, for example if a branch mispredict occurs. These requirements could be relaxed with more complex tracking, but a large number of common patterns are already covered by the implementation as is, and a simple compiler transformation can help to make others compliant (and we can expect the transform to be cheap on an out-of-order CPU, due to instruction-level parallelism).

5.1.6 Classifying loads, stores and memory locations

Classifications

Based on such recognised update sequences, let us classify each load instruction, store instruction and memory location into categories that will help with tracking reductions. Instructions are divided into (possible) *reduction* and *non-reduction* categories (e.g. reduction loads and non-reduction loads), whereas memory locations are categorised as *not accessed*, *loaded*, *reduction* or *non-reduction*. These classifications are inter-linked, as described below.

The classification of memory operations facilitates relaxed conflict checking, while the different memory states help with updating values correctly. *Loaded* and *reduction* memory locations are treated as reduction locations (so far). *Not accessed* locations do not matter, as they are not in the read or write sets, and do not have a local value, and *non-reduction* locations are treated as regular memory, subject to full conflict checking.

A load instruction is classified as a reduction if it forms part of a recognised reduction update sequence, and it targets a *not accessed* or *reduction* memory location. Furthermore, if the memory location is a *reduction* location, then the operation in the update sequence has to match the (previous) operation associated with the memory location.

We class the store instruction in an update sequence as a *reduction* store if and only if the target (which is necessarily the same as the one read by the load) is in the *loaded* state.

All other loads and stores are classed as *non-reduction* operations. Note that this means that any overlapping update sequences targeting the same location are classed as *non-reductions*, alongside accesses that are not part of a recognised reduction update sequence, as well as those targeting a memory location with an inconsistent reduction operation from the one encountered earlier for the same location.

State transitions for memory locations

The transitions for memory locations are shown in Figure 5.1. Any memory location targeted by a *non-reduction* memory operation at any point (whether it be a load or a store) is immediately classed as *non-reduction*. Memory locations start off as *not accessed*. A *not accessed* memory location targeted by a *reduction* load transitions to the *loaded* state, encoding the specific operation. Before the matching store is processed, any other loads are automatically non-reductions as per the above definition, eliminating overlapping update sequences. The only instruction that can maintain the reduction status of this location is the matching *reduction* store, which takes the memory to the *reduction* state. We still keep note of the operation performed. Now any loads that have a different operation or not part of a

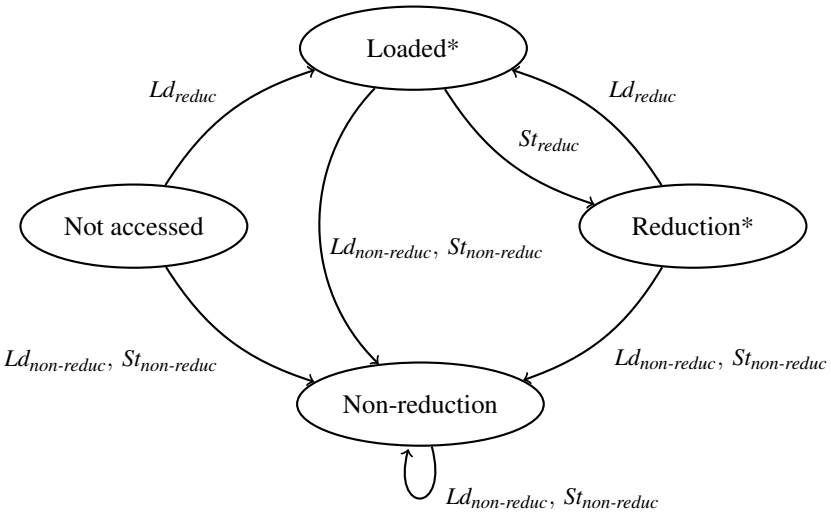


Figure 5.1: State transition diagram for memory locations. *Ld* means load and *St* means store. The edges not shown never happen due to the definitions of reduction loads and stores. The starred states store a specific reduction operation as well.

reduction update sequence, and any stores take us to the *non-reduction* state. However, a *reduction* load (which is necessarily part of an update sequence with the same operation, by definition) takes us back to the *loaded* state, where we wait for the matching store, just as before.

Implementation of transitions

The microarchitecture computes the above classification by observing the stream of instructions in register rename, and again in instruction commit. The protocol itself is run in instruction commit, but update sequences are constructed with the help of information from register rename. Both of these places observe the stream in order, and provide different advantages. Register rename gives an opportunity to find later elements of the update sequence before committing the start (e.g. seeing the update and store before committing the load, or seeing the kill before committing the store). On the other hand, observations in commit allow the microarchitecture to check the state of the memory location targeted by the update sequence (in Figure 5.1), as the final, translated physical address is known. This helps to work out the classification of the load and the store as *reduction* or *non-reduction*, as described above.

5.1.7 Conflict checking

In order to handle reductions without squashing, we need to ignore all conflicts, including true (read-after-write) conflicts, if they happen on a reduction location. Instead, the reduction handling logic will take care of hazards, updating values as needed.

To achieve this, *reduction* loads are not added to the epoch's read set, and they do not trigger conflict checking. On the other hand, *reduction* stores are still added as writes, and

they trigger conflict checking. Furthermore, *reduction* stores create specially tagged granules, which are incapable of *hiding* writes that occur later in older epochs.

This arrangement eliminates true conflicts on a memory location M between two epochs E and F ($E < F$) if M is recognised as a reduction (i.e. in the *reduction* or *loaded* state) in epoch F. If F has M as a reduction, and the epoch G (where $E < F < G$) has M in the *non-reduction* state, then a true dependence can occur between F and G (as usual), but it can also occur between E and G, despite F having written the value. This is because the write in E will cause the value in F to change, and this change is observable by G. If – instead – G had M as a reduction, then its own copy could also be updated when the write in E happens. If this is not the case, then the conflict between the update in F (triggered by a write in E) and the read in G is just like any other true read-after-write conflict, and thus G needs to be squashed.

An interesting detail here is that – in any of these cases – the status of M in epoch E does not matter. That is, the update could be a recognised reduction update or any other write operation. It does not matter which one it is, since the value in F can be updated, as we know how the current value was derived from the input value via a reduction, and the only instructions that observed the (now-stale) input value are the reduction updates, which have no observable side-effects that depend on the value.

5.1.8 Merging updates

In addition to relaxing conflict checking, eliding a true read-after-write memory conflict on a reduction location also requires merging the conflicting updates together. These updates are necessary in multiple different situations, as described in the next subsection, but the fundamental problem is the same each time, and can be formulated as follows.

We know a region of code (e.g. an epoch or a single update sequence) started with the value X, and obtained Y, but – meanwhile – some prior code (e.g. a previous epoch) changed the starting value from X to X'. We need to work out the new final value Y', thus merging together the updates $X \to X'$ and $X \to Y$.

We know Y was obtained using the reduction operator (say $\oplus$), by incorporating (e.g. adding) some value Z to X, i.e.

$$Y = X \oplus Z. \tag{5.1}$$

Then, the up-to-date, merged value should be

$$Y' = X' \oplus Z. \tag{5.2}$$

Unfortunately, we do not have the value of Z. The method for merging will be different for the different reduction operators. Specifically, we will divide into two sub-classes, depending on whether we can compute inverse elements under $\oplus$.

Operators with inverse

Under addition/subtraction and bitwise xor, each value has a well-defined inverse. That is,

$$\forall x.\ \exists x^{\ominus}.\ \forall y.\ (x^{\ominus} \oplus x) \oplus y = y. \tag{5.3}$$

Using this, we can compute the correct updated value Y' easily as

$$Y' = X' \oplus X^{\ominus} \oplus Y \qquad \textit{using eqs. (5.1) to (5.3).} \tag{5.4}$$

Operators without an inverse

For the other bitwise logical operators (i.e. bitwise and, bitwise or) and minimum/maximum, the inverse element is not well-defined. However, helpfully, these operators are idempotent. That is,

$$\forall x.\ x \oplus x = x. \tag{5.5}$$

With this property (and commutativity and associativity), notice that it is possible to merge two *well-behaved* updates (i.e. ones obtained by applying $\oplus$). That is, formally, suppose that

$$X' = X \oplus A \tag{5.6}$$

for some A. We already know that the update $X \to Y$ is well-behaved using eq. (5.1).

With this additional assumption, we can write

$$\begin{aligned}
Y' &= X' \oplus Z && \textit{by eq. (5.2)} && (5.7)\\
&= X \oplus A \oplus Z && \textit{by eq. (5.6)} && (5.8)\\
&= X \oplus X \oplus A \oplus Z && \textit{by idempotency} && (5.9)\\
&= (X + A) \oplus (X + Z) && \textit{by associativity and commutativity} && (5.10)\\
&= X' \oplus Y && \textit{by eqs. (5.1) and (5.6).} && (5.11)
\end{aligned}$$

This update can be easily calculated using $\oplus$, and is similar to eq. (5.4) (setting $X^{\ominus}$ to the unit value).

However, if the change is not *well-behaved* (i.e. if there is no A satisfying eq. (5.6)), then this identity does not hold, and in general, there is no way to perform the update. Thus, we need to check if the old update ($X \to X'$) was well behaved. Intuitively, we can easily do this for each operator. For example, for maximum, the input value into the epoch can increase but not decrease, and for bitwise or, setting additional bits is okay, but wiping bits is not.

Due to idempotency, we can rewrite this into a different form, as

$$(\exists z\ .\ x \oplus z = y) \iff x \oplus y = y. \tag{5.12}$$

The forwards implication can be proven as follows:

$$\begin{aligned}
& x \oplus z = y && && (5.13)\\
& \implies x \oplus x \oplus z = x \oplus y && \textit{by left-adding } x \textit{ under } \oplus \textit{ to both sides} && (5.14)\\
& \implies x \oplus z = x \oplus y && \textit{by idempotency} && (5.15)\\
& \implies y = x \oplus y && \textit{by applying eq. (5.13) to the left-hand side,} && (5.16)
\end{aligned}$$

and the reverse direction is trivial by setting $z \triangleq y$ as the witness.

This is helpful, as the right-hand side condition is much easier to check, using $\oplus$ and an equality check. In particular, we can check if the update $X \to X'$ is well-behaved by testing

$$X \oplus X' = X'. \tag{5.17}$$

If it is, then we can merge using this method. If it is not, then merging cannot be performed, and this implementation is forced to trigger a conflict, just like it would for any non-reduction reads that observed a stale value X.

I discuss this limitation in more detail in Section 5.1.10, as well as proposing alternatives. The update model described here is able to handle all pure reduction locations, and it also supports partial reductions. Additional squashing only becomes a problem if the reduction operation is not invertible, and non-reduction updates are frequent, and not well-behaved (according to the above definition).

Update algorithm In summary, we can always merge two updates $X \to X'$ and $X \to Y$ if there is an inverse element under $\oplus$. Otherwise, $\oplus$ is idempotent, and we can merge the updates so long as $X \to X'$ is *well-behaved*, in the sense that it could have been obtained by adding some value A to X using $\oplus$. The algorithm is shown in algorithm 5.

5.1.9 Detailed implementation

I describe the more detailed implementation of tracking reductions, updating values in the SSB, and recognising reduction updates by sketching out the pipeline structures that hold reduction metadata, and explaining how each one is updated and used.

The memory reduction table

The memory reduction table (MRT) is a cache that can hold information about a certain number of reductions from each epoch (limited by the size of the structure). This structure may be updated on every committed load or store instruction, as well as any forwarded (i.e. not hidden) stores from past epochs, as detailed below. The table is partitioned between epochs, and each slice works independently.

Algorithm 5 Merging two updates.

```
1: function MERGE(Operator ⊕, Update X → X′, Update X → Y)
2:     if INVERSE_⊕(X) ↓    // If inverse is well-defined, i.e. ⊕ ∈ {±, xor}.
3:         X_inv ← INVERSE_⊕(X)
4:         return X′ ⊕ X_inv ⊕ Y    // Calculate result using eq. (5.4).
5:     assert IDEMPOTENT(⊕)    // Otherwise, ⊕ ∈ {min, max, &, |}, so ⊕ is idempotent.
6:     if X ⊕ X′ ≠ X′    // Check if X → X′ is well-behaved using eq. (5.17).
7:         // It is not well-behaved, thus we cannot merge the updates.
8:         return ⊥    // Result is undefined.
9:     return X′ ⊕ Y    // Calculate result using eq. (5.11).
```

Entries Entries in the MRT table are of the form

$$(E,A) \mapsto (\text{Size},\ \oplus,\ V,\ V_{ld},\ \text{State}), \tag{5.18}$$

where E identifies the epoch the entry belongs to, A is the memory address of the variable (normally) storing the reduction, Size is the size of the variable (e.g. 32 bits or 64 bits), $\oplus$ is the reduction operation, V is the most up-to-date value of the reduction (in memory), V_{ld} is the value loaded by the latest load, which is used to correctly perform updates applied to *loaded* locations, and State is the state (according to Figure 5.1), specifically either *loaded* or *reduction*.

Determining the classification of a memory location The MRT and the read and write sets (in the conflict checker) together determine the reduction status of a memory location. If none of the granules are in the read or write sets and no MRT entry exists, then the state is *not accessed*. If there is an MRT entry, then its State field gives the state, as either *loaded* or *reduction*. Otherwise, the state is *non-reduction*.

Handling reduction loads When the first reduction load targeting a given location (in a given epoch) is committed, a memory reduction table entry is created, containing the size, operator, and the value read by the load (V_{ld}). V is determined by performing a read in the SSB, as this could be different from V_{ld} due to stores performed by older epochs between executing and committing the load. The state is set to *loaded*. On successive *reduction* loads, only the state is updated and V_{ld} is set.

The current method works (unmodified) when a reduction load gets a value forwarded from a previous (reduction) store from the same epoch via the load-store queue (or the store buffer). In these cases, it is possible that V_{ld} is already out-of-date when the load is executed (if the value in the SSB and the MRT got updated from an older epoch), but this is corrected by the standard merging logic – without requiring extra attention – when the matching reduction store is committed (as this will look like just a bigger, combined update).

Handling non-reduction loads Upon committing a non-reduction load, matching MRT entries from the same epoch are invalidated.

Handling non-reduction stores Similarly, a non-reduction store invalidates matching MRT entries from the same epoch. Furthermore, if the MRT entry was in the *loaded* state, then we retrospectively perform conflict checking on the already-committed load instruction of the in-flight update sequence. To do this, we compare V_{ld} with V in the MRT entry. If they match, then the load observed correct data. Its granules are added to the read set (as we can no longer update its starting value), and execution continues. If $V_{ld} \neq V$, then the load observed stale data, but it has not yet been updated. A more complex implementation could attempt to update the associated register in the pipeline (and squash any users if needed), but the prototype avoids these complications by simply squashing the threadlet if this happens, as this should be a relatively rare case (since there needs to be an in-flight reduction update, and the reduction needs to be imperfect).

Algorithm 6 Merging a write update with younger reduction updates.

Require: R_E, W_E: the read and write sets of epoch E
Require: $\text{MRT}(E, M)$: the memory reduction table entry for epoch E location M.

```
1:  function WRITE(Epoch E, Location M, Update V_old → V_new)
2:      for all e from E + 1 to YOUNGESTACTIVEEPOCH()
3:          assert M ∉ R_e    // Conflict detection would have caught non-reduction reads.
4:          if MRT(e, M) ↓    // If M is a reduction in e, merge updates.
5:              (Size, ⊕, V, V_ld, State) ← MRT(e, M)
6:              V_merged ← MERGE(⊕, V_old → V_new, V_old → V)
7:              if V_merged ↑
8:                  // If the updates cannot be merged, this is an unresolvable true conflict.
9:                  CONFLICT(e)
10:                 return
11:             else
12:                 // Update the value V in the MRT entry with the new value, V_merged.
13:                 MRT(e, M) ← (Size, ⊕, V_merged, V_ld, State)
14:                 // By doing so, we updated V → V_merged in epoch e.
15:                 // Hence, future epochs need to see the update V → V_merged.
16:                 V_old ← V
17:                 V_new ← V_merged
18:         else
19:             if M ∈ W_e
20:                 // There was a (hiding) non-reduction write in e.
21:                 // Thus, younger epochs cannot observe the current update.
22:                 return
```

Lastly, we add the store to the write set, perform conflict checking from it (to check if any non-reduction reads from younger epochs observed the change), and update any reductions from younger epochs that should have observed this write. The algorithm for this is shown in algorithm 6. We start with E, the store's epoch, its target location M, and the update it performed, $V_{old} \to V_{new}$, where V_{old} is the value overwritten by the store in memory (in the SSB), and V_{new} is the (newly) stored value. Instead of looking this up, we could store the starting value in the next epoch's MRT entry (i.e. add a field V_0 to each entry, storing the up-to-date value forwarded in to the entry). The algorithm iterates through epochs, looking for MRT entries, and *hiding* non-reduction writes, and updates any *non-hidden* MRT entries using the MERGE function from algorithm 5.

Handling reduction stores The matching *reduction* store updates the state from *loaded* to *reduction*, and also compares the V_{ld} with V. Since the value loaded, V_{ld}, was the most up-to-date value at the time the load was executed, it will equal V, unless V was updated during the lifetime of the update sequence. If this is the case, then the value contained in the store (say V_S) is stale. In particular, we need to merge the update $V_{ld} \to V$ that happened in memory, with the update $V_{ld} \to V_S$ performed by the update sequence. This can be done by calling MERGE from algorithm 5.

Irrespective of whether we had to merge an update from a previous epoch or not, we also need to propagate the update to younger epochs. Firstly, the store is added to the write set, and conflict checking happens as usual, in order to detect conflicts with regular, non-reduction reads from younger epochs. Secondly, the WRITE method from algorithm 6 is invoked, in order to merge the current update with reductions from younger epochs that observe it.

The rename reduction table

Apart from tracking values and reduction locations in memory, we also need to track update sequences in the pipeline. The rename reduction table (RRT) is a structure in the register rename stage, which tracks the status of a small number of in-flight sequences that could be reduction updates based on recent instructions seen by the register rename stage.

Entries Entries in the RRT are of the form

$$(\oplus,\ A_{enc},\ R_{ld},\ R_{upd}). \tag{5.19}$$

Here, $\oplus$ is either a reduction operator, or the special value `UNKNOWN`. A_{enc} is the encoded memory address. That is, an expression of the form `[p132 + #100]`, where `p132` is a physical register, and `#100` is an immediate constant (the offset). R_{ld} is the (physical) register written by the load instruction, and R_{upd} is the physical register written by the update (if the update has been renamed already).

The RRT is content-addressable, lookups can be performed either based on R_{ld} or R_{upd}. This structure only contains a few entries, as an entry only needs to be allocated on a load instruction, and it only needs to live for a few cycles of rename activity, until the corresponding update and store instruction are renamed, and the physical register mappings are overwritten. After this, the information here moves to the in-flight reduction table, as described later.

Updating the RRT On a load, a new entry is allocated, setting the $\oplus$ to `UNKNOWN`, and setting A_{enc} as well as R_{ld}. R_{upd} remains unset until the update is seen. The update instruction is recognised by pattern-matching (e.g. `op x, x, y`, where x is the architectural register mapping to an R_{ld} in the RRT). The update sets the R_{upd}. Following this, the we look for a store instruction reading R_{upd} and using the address A_{enc}. Matching the physical register and offset in A_{enc} ensures that the two addresses are the same, due to physical registers being static (single-use). When the store is found, an entry is created in the in-flight reduction table (as described below), mapped to from both R_{ld} and R_{upd}. The entry in the RRT can then be recycled, and we the kill instruction is identified with the help of the IRT (as any instruction overwriting a R_{upd} variable).

Removing failed entries If – during this process – any other instructions observe R_{ld} or R_{upd}, then the entries in the RRT (and in-flight reduction table, if the entry has already been created) are destroyed, which causes the back end pipeline to treat the associated instructions

as non-reductions. If the load instruction is committed before the kill instruction is found, or squashed at any time, then the entries are likewise erased. If the pipeline is squashed such that some but not all of these instructions are squashed, then the entries can be repaired in most cases (e.g. if the update instruction is squashed, then R_{upd} needs to be reset), but an easy approach is to just discard the entries.

Care needs to be taken to ensure that a pipeline squash cannot lead to a load instruction being treated as a reduction by the conflict detector, but then there being no corresponding update, store and kill. The prototype opts for a simple implementation that also discards the mappings if a conditional or indirect branch is encountered – which could result in redirection of control flow – is seen between the load and the kill, and similarly on any barrier instructions. This way, any pipeline squash will only target the full reduction update sequence, or none of it.[5]

The in-flight reduction table

The in-flight reduction table (IRT) is a structure tracking in-flight reduction update sequences in the pipeline. In particular, we add an IRT index field to the metadata of each live physical register. This index points to an IRT entry, which holds some data about the reduction update sequence containing the instruction that writes to that physical register. In particular, the physical register R_{ld} and R_{upd} will point to the same IRT entry, containing data about the sequence.

Entries Entries in the IRT are of the form:

$$(\oplus,\ R_{ld},\ R_{upd},\ A,\ \text{Status}), \tag{5.20}$$

where $\oplus$ is the (assumed) reduction operation, R_{ld} and R_{upd} are the physical registers written by the load and the update (respectively), as before, A is the value of the memory address (if already known), and Status describes the progress of the reduction through the pipeline. In particular, Status will go through the following states: *store-renamed* $\rightarrow$ *kill-renamed* $\rightarrow$ *load-committed* $\rightarrow$ *update-committed* $\rightarrow$ *store-committed*, before the entry is recycled.

Updating the IRT Mappings here are created when the store instruction is renamed. At this point, the entry is set to $(\oplus,\ R_{ld},\ R_{upd},\ \perp,\ \textit{store-renamed})$.

If the load were to commit now, it would destroy the mapping here, and it would be treated as a non-reduction load, as we do not know if a kill instruction will or will not exist. Once a suitable kill instruction is renamed, the rename stage looks up the mapping from the overwritten physical register into the IRT, and updates the Status to *kill-renamed*.

Then, when the load is committed, we fill in the actual physical memory address A in the table, and look it up in the MRT, and the epoch's read and write sets to determine the classification of the memory (i.e. the state in the transition diagram in Figure 5.1), and the associated reduction operation $\oplus_{MRT}$ (if one exists). This is used to classify the load as

[5] Except memory order violations, exceptions and context switches. In these cases, we can squash the threadlet if it is speculative, otherwise discard the mappings and fix up register state if necessary, and run non-speculatively. This is costly, but very infrequent.

reduction or *non-reduction*, as described in Section 5.1.6. The load then updates the MRT entry, as appropriate, based on its classification (as described previously). A *reduction* load also sets the IRT Status to *load-committed*, while a *non-reduction* load discards the IRT entry.

Following this, the update will be committed, which updates the *Status*, destroys the physical register R_{ld}, together with its link to the IRT (but the IRT entry is kept).

Finally, the store and the kill are committed, which are recognised as *reduction* operations with the help of the IRT and MRT entries. Normally, the store will only update the Status, while the kill will recycle the IRT entry. However, in the edge case where the pipeline gets squashed between the reduction load and store commit, the IRT entry is used to fix up state (this is an unexpected rare case, therefore the sub-optimal performance implications are acceptable).

5.1.10 Limitations

The above design handles in-memory reductions on scalar and array values, so long as the reduction uses one of the recognised reduction operators (i.e. integer addition/subtraction, bitwise logical operators, minimum, maximum), and updates are performed in the recognised form (containing a simple load-update-store-kill pattern with no intervening branches and reads, matching the required address encoding, and utilising a scalar register). Furthermore, it also elides conflicts from non-reduction (or unrecognised) stores to reduction locations, provided the update is well-behaved (i.e. the change in value could have been obtained by applying the reduction operator) for non-invertible operators (e.g. bitwise logical operators, minimum and maximum).

Most obviously, we could expand coverage to slightly more complex update sequences, not containing a kill instruction, or containing intervening branches, as described previously. This would make tracking more difficult, and also require a robust register update method in case the register used for the reduction turns out to be still live after the reduction update pattern is done.

Additionally, vectorised reduction updates can be supported too, with similar tracking. This is possible irrespective of where the values are loaded (and stored) from, whether they be arranged in a contiguous, strided, or general (scatter-gather) pattern.

The requirement for invertible or idempotent operators with well-behaved updates comes from the fact that we default to executing reduction updates as-is, on the most up-to-date value read from the SSB at the time the load instruction is executed. Instead, the load could be provided with the unit value for the reduction operator, thus computing a per-epoch private sub-total for each reduction location, which could be added on to the up-to-date input value a single time, once the predecessor has performed all of its updates (e.g. when the current epoch commits). This allows arbitrary updates from past epochs (e.g. re-setting a bitwise-or reduction location to 0, wiping bits). Furthermore, the implementation footprint, and the contention on reduction ALUs may be lower in this case for frequently-updated locations. Specifically, the implementation proposed earlier in this section may require up to $E \cdot N \cdot T$ updates – where E is the number of epochs in the loop, N is the number of updates per epoch and T is the number of threadlet contexts in the microarchitecture – since each update

could change the value in each active younger threadlet. In the private-value version, we only require E merge operations. On the other hand, for uncontended reductions (e.g. for sparse array reductions), the original implementation requires very few or no merges at all (since the values computed in the pipeline are correct), while the private-value version still requires a merge operation for every location updated in every epoch. A more sophisticated implementation could use run-time heuristics to select between the two approaches.

Finally, the prototype itself also has some limitations. Firstly, contention on the reduction tables (i.e. RRT, IRT, MRT) or the reduction ALUs is not modelled. For tables, the size is not limited either. Furthermore, it is assumed that merging values can be done in the background, in parallel with other work, thus no cycles are added for this.

5.1.11 Case study/evaluation

I evaluated the approach on the GAP benchmark suite, as it shows clear examples, and the code size of the benchmarks make this suite suitable for manual analysis and annotation.

I used the five original, fully sized input graphs (0.5 to 32 GiB in size) to generate SimPoints, but reduced the number of repetitions to make simulation times reasonable.[6] I manually annotated the most promising loops, eliminating those that did not produce a speedup. For this experiment, I did not simulate a limit on the size of the structures, as we expect the number of reductions tracked to be low by the nature of the benchmarks.

Figure 5.2 shows the achieved speedups by benchmark (taking the geometric mean of run times between different input graphs) with and without reductions. Without handling reductions specially, the scheme achieves a 9.8% geometric mean speedup over the whole suite, with a maximum benchmark speedup of 38% (although individual benchmark-graph pairs show a wider range of speedups, up to 3.1×). With reduction handling added, the geometric mean speedup increases to 17.0%, with the largest benchmark speedup of 2.1× being observed for *tc*. The increases are driven by *tc* (52.4% improvement) and *bfs* (8.8% speedup). Unfortunately, the loop containing reductions identified in *bc* remains unprofitable, even with reductions handled well, due to resource saturation and iteration size.

[6] Most of the workloads in GAP run the algorithm 16 to 64 times, which was reduced to 1 to 4 repetitions in this case, to limit checkpoint generation time.

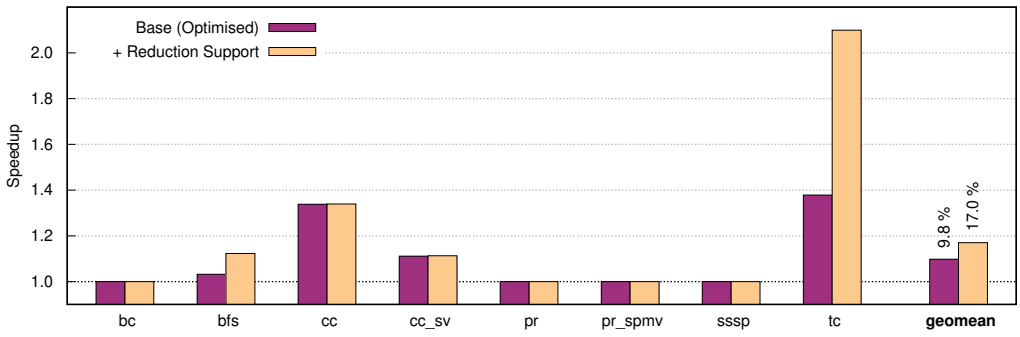


Figure 5.2: Speedups for the GAP benchmark suite. The base configuration includes eager forwarding and iteration packing (from Chapter 4). It squashes threadlets upon any true dependence violations. The improved scheme merges reduction updates without squashing.

This shows the benefits of handling reductions well for specific loops (such as those in *tc* and *bfs*), although we can also see that benchmarks may have other bottlenecks that limit performance. For a number of the regions analysed here, conflicts arising from the reductions masked other sources of conflicts, therefore it was important to eliminate reductions in order to reveal a whole set of underlying issues.

5.2 General register dependences

5.2.1 Motivation

Cross-iteration register dependences exist in almost all loops. So far, the architecture limited true register dependences to the header and the continuation for successful parallelisation, and – crucially – register values created in the body that got read in any future section of the loop (continuation, header or body) resulted in a conflict and squash just before the reading epoch could commit. This restriction can be enforced by the compiler by inserting the detach and reattach instructions either both below or both above the register update that created the cross-iteration dependence. This tends to work well for simple induction variables, as they tend to be naturally updated either near the top or near the bottom of the loop iteration, thus most of the iteration can still form part of the body.

Complications arise when this assumption does not hold. In some cases, simple compiler transformations could help to increase the size of the body. Code motion could be used in some cases to move the cross-iteration register dependence up to the header or down to the body, such as in the example shown in Figure 5.3.

However, such transformations can become difficult for multiple possible reasons. Firstly, data or control-flow dependences between the register update and the rest of the loop can limit movement. If the update is dependent on prior code, then the prior code needs to be moved or copied together with the dependent register update. If we move it up, then this is to preserve their order, and if we move it down, then it is to eliminate data-flow between the body and the continuation (where the register update now resides). Copying may be necessary if other values in the loop also depend on the same values, leading to code bloat. A common special case of this is when we have a register update on a conditional path (or inside an inner loop), which also contains other code. Secondly, it may not be possible to safely perform the necessary code motion at compile time if the code contains memory accesses that may alias with the rest of the loop, or if calls to non-inlined functions are involved (which could contain may-alias memory accesses).

5.2.2 Infrequent register dependences from the body

The appropriate handling for the remaining dependences in the body depends on their frequency. If they are sufficiently infrequent, the compiler can simply emit code with the dependence left in. These will be caught by the register conflict checking (described in Section 3.2.4), and handled similarly to in-memory cross-iteration dependences. The only difference is that in-register conflicts are detected if the register value changes between the start and finish of the epoch instead of simply monitoring writes. This means that an epoch

```
1  for (int i=0; i<N; i++) {
2      foo1(i);
3      foo2(i+1);
4  }
```

(a) Simple example loop in C++.

```
1       mov     w19, wzr
2       ldr     w20, [x20, :lo12:N]
3  .loop:
4       mov     w0, w19
5       bl      foo1(int)
6       add     w19, w19, #1
7       mov     w0, w19
8       bl      foo2(int)
9       cmp     w19, w20
10      b.lt    .loop
```

(b) Assembly (`clang-17 -O3 -march=armv8-a -S`).

```
1       mov     w19, wzr
2       ldr     w20, [x20, :lo12:N]
3  .loop:
4       detach  .cont
5       mov     w0, w19
6       bl      foo1(int)
7       reattach .cont
8  .cont:
9       add     w19, w19, #1
10      mov     w0, w19
11      bl      foo2(int)
12      cmp     w19, w20
13      b.lt    .loop
```

(c) Naïve hint insertion only parallelises `foo1`.

```
1       mov     w19, wzr
2       ldr     w20, [x20, :lo12:N]
3  .loop:
4       detach  .cont
5       mov     w0, w19
6       bl      foo1(int)
7       add     w0, w19, #1
8       bl      foo2(int)
9       reattach .cont
10 .cont:
11      add     w19, w19, #1
12      cmp     w19, w20
13      b.lt    .loop
```

(d) Optimal hint insertion captures both `foo1` and `foo2`.

Figure 5.3: In this loop, the compiler put the induction variable increment (in blue) between the calls to `foo1` and `foo2`. Since this increment needs to be in the header or the continuation, naïve hint insertion will only capture either `foo1` or `foo2` in the body, while the other one will run sequentially. Optimal insertion can include both, by moving the induction variable update down, and turning a `mov` instruction into an `add` to calculate $i+1$ before calling `foo2`. This value dies when the body ends.

can write to a register that is read by a successor and still not cause a conflict so long as the final value is the same as the starting value. Furthermore, it also means that conflict checking can only be performed as the epoch reaches the reattach point, rather than on-the-go, as with memory, leading to conflict detection – and thus restarting after a squash – happening later. Thus, we should not expect this method to work well for anything but very infrequent dependences (e.g. based on a rare condition check passing).

Hint support for improved performance

A possible improvement to this is to distinguish between different registers, or explicitly forward values or squash future epochs. In the first case, we can perform eager value forwarding and conflict checking on certain registers that contain infrequent cross-iteration dependences. The identification of such registers could be achieved using additional hints, inserted by the compiler, or by run-time monitoring. The downside is that any writes to these registers – including in nested functions, after the register has been spilled to the stack – will result in a conflict.

This could be tackled by explicitly forwarding values (and triggering conflict checking) when a genuine update is performed. The update instruction could be tagged, or a hint could be added afterwards, signifying that the current write to the given register is likely to alter the final value of the register at the end of the epoch. This approach effectively distinguishes between the two types of registers, as well as the two types of updates (i.e. updates to the same *variable* in the same register and updates to a different (local) variable that happens to be in the same register).

Lowering to memory

A lower-impact approach – which already works with the current hints – is to lower any variables that exhibit dependences to memory. The compiler can allocate a location on the stack, and spill the register before the start of the loop. Then, the variable needs to be re-loaded in each epoch that uses it, and stored back in each epoch that updated it.

There are three types of infrequent register dependences, depending on the frequency of reads and writes (updates), as each conflict requires a read and a write from close-by epochs. We can have infrequent writes and infrequent reads, infrequent writes but frequent reads, or frequent writes but infrequent reads. If both reads and writes are frequent, then dependences would also be frequent.

Depending on this classification, and the number of uses along different code paths, the variable can be loaded and stored either at the start (and end) of the epoch, or close to each (independent) use. The choice can be optimised to keep the conflict frequency low even after lowering. For example, if the variable is only accessed for a read-modify-write operation along an infrequent path, then it can just be loaded, modified and stored back in that path. On the other hand, if the variable is read in each epoch, but only infrequently updated, then it can be loaded at the start of the epoch (e.g. in the continuation), but only stored to whenever an update is performed (on an infrequent path), thus only triggering a conflict on the infrequent updates and not on the frequent reads. Similarly, variables that are frequently written (but

infrequently read) can be stored back just before the reattach in each epoch, but only read on-demand.

Note that the expressive power of lowering to memory is the same as that of the hint support shown above. Specifically, a store to memory is interpreted as an explicit forwarding operation, while a load from memory adds the variable to the read set (just like reading the register would). Thus, a load from memory enforces squashing if a write from a younger epoch is executed later.

The additional overhead of this approach is the run-time cost and code footprint of the extra load and store instructions. Since we expect stack frames to be cached, and either the loads, the stores, or both are infrequent, and the total number of such registers is expected to be low (since the number of registers in total is already limited, and many will hold induction variables and temporaries), we can expect this overhead to be relatively low compared to the size of an epoch and other costs associated with launching and tearing down threadlets.

Conclusion

Thus, we already have a mechanism for handling infrequent true register conflicts in the architecture, conditional on compiler support. Hence, from an architectural and microarchitectural point of view, the more interesting set of register dependences is frequent dependences. I discuss handling these for the rest of this subsection.

5.2.3 Considering frequent true register dependences

For frequent true register dependences, relying on conflict checking and squashing is sub-optimal, because most epochs will be squashed. Instead, we would like to avoid dependence violations by forwarding values once they have been produced, and waiting for incoming forwarded values before consuming them.

Register dependences are particularly suitable for implementing such forwarding for a few reasons. Firstly, registers show up explicitly in the compiler's intermediate representation (IR). Since the variable stored in the register is live (as it causes a dependence) between iterations, it will show up as a (scalar) variable (i.e. collection of update instructions and ϕ-nodes), which makes it feasible to find and analyse the dependence in the compiler (although this has not been implemented in the prototype).

Secondly, an out-of-order processor pipeline provides a good mechanism for stalling instructions that (directly or indirectly) depend on the current value of a specific register. Namely, a (physical) register can be marked as 'not ready' in the register scoreboard, achieving this, while permitting any independent instructions to execute early, out-of-order. Furthermore, control dependences from the blocked register are elided using branch prediction, allowing for more independent instructions to be discovered if the branch outcome can be predicted with good accuracy.

The next sections describe hint instructions and a microarchitectural implementation for handling frequent cross-iteration register dependences. The architecture described was devised and refined in collaboration with colleagues from Arm.

5.2.4 Hint semantics

Let us introduce two new hints, to enable such synchronisation, called *send* (`send`) and *receive* (`recv`), each taking two arguments. The first argument is a register, which exhibits a frequent true dependence, and the second argument is a region ID. As with other hints, the region ID is the address of the continuation, and it is used to check if the hint corresponds to the current active region (i.e. the region we are currently parallelising). If the region ID does not match, the hint is simply ignored. If it does match, then the hints follow the semantics outlined below.

As with other hints, send and receive do not affect sequential semantics, but they encode some parallel semantics, which can be used by the microarchitecture to perform more efficient parallelisation. Thus, the microarchitecture is still responsible for checking and maintaining sequential semantics regardless of the hints. Specifically, true register conflicts not correctly guarded by these hints should result in the default behaviour of either fixing up values (if possible), or squashing conflicting younger epochs. As with the original hints, having the hints inserted by the compiler communicates useful information to the microarchitecture, thus helping it gain performance in the expected case (i.e. if the registers identified conflict frequently, but only in a way compatible with the parallel semantics described below). Having the hints simplifies the tracking logic in the hardware, thus enabling frequent conflicts to be handled well at a reasonable tracking cost.

Simple parallel semantics

Receive The instruction `recv` R_x`,` C signals that the architectural register R_x is involved in a frequent, true, cross-iteration register dependence in region C. Furthermore, its placement signals the synchronisation point. Specifically, the hint suggests that any accesses to R_x after the *receive* depend on the up-to-date input value from the previous epoch, which will only be available when the previous epoch has executed `send` R_x`,` C, or it has reattached (i.e. committed `reattach` C). If either of these events has already happened, then receive can proceed immediately, without blocking.

Send The instruction `send` R_x`,` C signals that the register R_x is involved in a frequent, true, cross-iteration dependence in region C, and the compiler expects that R_x has reached its final value for the current epoch, and thus the register value is ready to be forwarded to the next epoch. The hint suggests that after forwarding the current value, any `recv` R_x, C instructions in the successor epoch may be unblocked. If the successor has reached `reattach`, then values should also propagate through to the next epoch.

Usage examples

These two hints can be used in multiple different ways, with two examples shown in Listings 5.3 and 5.4. The `filter` function filters elements[7] based on a complicated predicate, and exhibits a general read-modify-write dependence on the variable `output_size`,

[7] The pattern here uses an output array with a size variable, for clarity. The same pattern also works with the C++ `std::vector` type, as `push_back` gets inlined, and the vector size is held in a register.

Listing 5.3: Loop with a true cross-iteration register dependence on `output_size`.

```
// Copy elements that pass complicated_test from input to output.
// Contains conditional read-modify-write dependence on output_size.
void filter(vector<Item> input, Item output[], int& output_size) {
    output_size = 0;
    for (const Item& x : input) {
        if (complicated_test(x)) {
            // recv(output_size);
            int index = output_size++;
            // send(output_size);
            output[index] = x;
        }
    }

    // Results are in output[0 .. output_size-1];
}
```

Listing 5.4: Loop with a true cross-iteration register dependence on `savings`.

```
// Calculate total savings with interest after a number of years.
// Contains non-reduction accumulation on savings.
double calculateSavings(int years) {
    double savings = 0;
    for (int i=0; i<years; i++) {
        double new_savings = SavingsInYear(i);
        double interest_rate = InterestRateInYear(i);

        // recv(savings);
        savings = (savings * (1.0 + interest_rate)) + new_savings;
        // send(savings);
    }

    return savings;
}
```

while `calculateSavings` shows a non-reduction accumulation, where the update (multiply-accumulate) is not associative.

The anticipated hint placements are shown in comments. Note that the actual hints act on registers, and also carry a loop ID. The annotations here only help to logically show the workings of the scheme. Furthermore, I syntactically laid out the code in a way that helps hint placement (e.g. `output_size` is saved and updated first, then only used on the next line). A programmer can easily make such changes, and the exact syntax is abstracted away in compiler IR, where automated insertion would occur.

Without the new hints, we would need to move the update to either the header or the continuation. In either case, the whole dependence chain would also need to move (for ordering and eliminating dependences from the body to the continuation), leaving only very little or nothing in the body. In the `filter` example, only the store to the `output` array can remain in the body, while in the second example, the body would be empty.

Note that the `send` at the end is not necessary, as `reattach` would automatically have the same effect. Similarly, the fact that both hints are on a conditional path in `filter` does not matter, as `reattach` will forward the unmodified value.

Edge cases for parallel semantics

While in the examples above, the two hints come in pairs, neatly forming a critical section where the value is updated, this need not be the case. Furthermore, the examples do not have any accesses to the registers outside of the critical section. There are a few edge cases that can be handled specially.

Accesses before receiving There may be accesses to R_x before the receive hint. If this is due to imprecise hint insertion, then the register read and write set checks of the microarchitecture are expected to catch such behaviour. Otherwise, the accesses may be due to special patterns, such as register spill and fill. Previously, these accesses were specially handled, and spill slots in memory were patched up. We can do the same in the presence of the new hints too, patching up memory when the final value is forwarded.

Other write accesses (i.e. those not handled specially, e.g. by spill and fill support) before receiving a value also need to be monitored now. Any writes to the register will overwrite (and kill) the current value. Thus, if a receive is placed after a write in the same epoch, that write produced a newer value, and so the forwarded value is no longer live. Therefore, the receive must not alter the register's value, and it can be treated as a no-op. A microarchitecture that cannot handle this case must still detect a violation and squash if it occurs, in order to maintain sequential semantics.

Accesses after sending Read accesses after the send hint do not cause a problem, as the up-to-date value is live and available. However, writes may pose an issue. Specifically, the region after the send should be treated like any parallel body that does not have send and receive hints. That is, the register value must be the same between the send and the end of the epoch, or else a violation must be logged, and squashing initiated.

Send without receive The send instruction means that the current value of the register is final. If the register is already in the epoch's write set, then this case is simple: the current (local) value of the register should be forwarded. For example, this pattern could occur in a program that increments a counter in some iterations (thus receives and sends it), but simply resets it to zero in others.

On the other hand, if the register has not been written to yet, then the current value of the register in the present epoch will depend on whether the predecessor has forwarded a value yet or not. Thus, it makes sense to define a single send instruction as an indication that the current epoch will not change the value of the given register, but it also does not require it (at present). This means that as soon as a value has been forwarded by the predecessor epoch, we can forward it straight through to the successor. Note that it could be the case that the current epoch later requires the value of the register, and in this case it could call recv to block and obtain it. This may not be a common pattern, but it is possible, as shown

Listing 5.5: Example where we receive after sending a value.

```
1  void exitCheck(int N) {
2      int sum = 0;
3      for (int i=0; i<N; i++) {
4          if (test(i)) {
5              // recv(sum);
6              sum += f(i);
7          }
8          // send(sum);
9
10         g(i);
11
12         // recv(sum);
13         if (sum > 10) break;
14     }
15 }
```

in Listing 5.5. Here, the value is conditionally updated first, after which point, it is final. If it has not been updated, then we only need to receive it near the end of the epoch, where it is used in a condition check.

5.2.5 Microarchitecture

Let us first devise a simple working implementation for taking these hints and performing synchronisation. The next section then investigates performance issues, optimisations, and inter-operability with iteration packing. For now, let us assume that each epoch contains a single iteration.

Algorithms 7 and 8 show how the new hints are handled in various stages of the pipeline. Specifically, it shows events in instruction decode, register rename, execute (and writeback), and instruction commit.

Some extra state is tracked, in order to handle the hints correctly. I expand the different bits of state below, briefly explaining how they are used.

Input value The InputValue(E,R) structure contains the starting, input value for each register for each (active) epoch. It also has a *valid* flag, which is set to *false* (InputValue$(E,R)\uparrow$) by default for newly launched epochs. Setting the InputValue marks the entry valid. When entering a new region, we set InputValue(E,R) to the current value of R. Afterwards, we set the value for each epoch as soon as the input value can be confirmed. This can happen when the current epoch signals that it will not modify (produce) the given register by committing a send hint, or reaching the reattach instruction (i.e. when reattach gets to the front of the ROB), or if the same happens in a past epoch, and the current epoch has already signalled that it does not intend to produce R (as before). The PROPAGATE method takes care of propagating values as far as possible, as shown in the algorithms.

Send bitmask For each epoch, we store the set of architectural registers Snd(E) that have been targeted by committed send hints in a given epoch E, as a bitmask. We also maintain the same bitmask in the decode stage, Snd$_D(E)$.

Algorithm 7 Handling recv hints in the pipeline.

```
1:  function DecodeRecv(Instruction I)
2:      if ¬MatchesCurrentRegionID(I)
3:          return Nop()
4:      E ← CurrentEpoch()
5:      R ← ExtractRegisterArgument(I)
6:      if R ∈ Wr_D(E)
7:          // Register written (e.g. already received or otherwise written to). Ignore hint.
8:          return Nop()
9:      // Add register to write set, just like on any other instruction that writes R would.
10:     Wr_D(E) ← Wr_D(E) ∪ {R}
11:
12:     return Recv(E, R)

13: function RenameRecv(Epoch E, Register R, Instruction I)
14:     // Grab a new register, recycle the old one.
15:     P_old ← RenameMap_E(R)
16:     P_new ← AllocatePhysicalRegister()
17:     Kill_I ← {P_old}
18:     Output_I ← {P_new}
19:     RenameMap_E(R) ← P_new
20:     Scoreboard(P_new) ← NotReady

21: function ReadyToIssueRecv(Epoch E, Register R)
22:     return (InputValue(E, R) ↓)    // Issue recv iff input value is available.

23: function ExecuteRecv(Epoch E, Register R, Instruction I)
24:     // We already checked that the input value is available, so just obtain it and write it to the register.
25:     V_in ← InputValue(E, R)
26:     {P_out} ← Output_I
27:     ValueOf(P_out) ← V_in
28:     Scoreboard(P_out) ← Ready    // Update scoreboard on writeback.

29: function CommitRecv(Epoch E, Register R, Instruction I)
30:     Wr(E) ← Wr(E) ∪ {R}    // Receive counts as a write.
```

Algorithm 8 Handling send hints in the pipeline.

```
1:  function DecodeSend(Epoch E, Register R, Instruction I)
2:      if R ∈ Snd_D(E)  // If we have already sent, this is a no-op.
3:          I ← Nop
4:          return
5:      Snd_D(E) ← Snd_D(E) ∪ {R}

6:  function RenameSend(Epoch E, Register R, Instruction I)
7:      P ← RenameMap_E(R)
8:      Input(I) ← {P_old}
9:      Kill(I) ← ∅
10:     Output(I) ← ∅

11: function CommitSend(Epoch E, Register R, Instruction I)
12:     Snd(E) ← Snd(E) ∪ {R}
13:     {P_in} ← Input(I)
14:     Propagate(E, R, P_in)    // See implementation below.

15: function ReattachAtFrontOfROB(Epoch E)
16:     // This can be done asynchronously, and in parallel.
17:     for all R ∈ ArchitecturalRegisters()
18:         if R ∉ Snd(E) ∧ (R ∈ Wr(E) ∨ (InputValue(E, R) ↓))
19:             // Value for register R is ready but has not been forwarded yet, so forward now.
20:             P ← CommitRenameMap_E(R)
21:             Propagate(E, R, P)
        // If done asynchronously, wait here until the loop completes.
22:     ReattachReached(E) ← ⊤

23: // Propagate value of R from epoch E to all (confirmed) consumer epochs.
24: // Any past (or potential future) writes from epoch e > E mean that all e' > e are not (confirmed) consumers.
25: function Propagate(Epoch E, Register R, PhysicalRegister P)
26:     Assert R ∈ Wr(E) ∨ (InputValue(E, R) ↓)    // Precondition: final value of R is available.
27:     if R ∈ Wr(E)
28:         V ← ValueOf(P_in)
29:     else
30:         V ← InputValue(E, R)
31:     for all e ← E + 1 to YoungestActiveEpoch()
32:         InputValue(e, R) ← V    // Propagate to epoch e.
33:         // Now check if we should forward on to later epochs.
34:         if R ∈ Wr(e)
35:             // Epoch e has produced R, thus hiding the output of E from younger epochs.
36:             // That is, any e' > e consume the value from e (or an even younger epoch), and not from E.
37:             return
38:         if ¬(R ∈ Snd(e) ∨ ReattachReached(e))
39:             // Epoch e may still produce R, so we do not know if e' > e will consume from E or e.
40:             return
41:         // Otherwise, e does never produces R, so we propagate our value through to e + 1.
```

We use Snd to forward values to the correct epochs at the correct times (e.g. to propagate values straight through epochs that have sent but not received), and Snd_D is used to de-duplicate repeated hints (e.g. to ignore the second send call for the same register). If the register is already in the send set, then the instruction is turned into a no-op. Hints carry this information in the pipeline, and thus we can easily reset the correct bits in the bitmask whenever hints get squashed due to branch mispredicts.

Register write mask Besides maintaining the register read set $\text{Rd}(E)$ per epoch, we also maintain the write set $\text{Wr}(E)$ as a bit mask. Register spills and fills are excluded from both sets. Note that this mask is different from the one used for general register conflict checking, as we normally use the change set $\text{Chg}(E)$, i.e. the set of registers that have changed value between the start and end of an epoch. The write set is used to determine the semantics of the send instruction, as outlined in the edge cases above. That is, if $R_x \in \text{Wr}(E)$, then we can forward a value to the next epoch straight away, whereas otherwise the value forwarded will come from the predecessor. The current value of the register in the current epoch is irrelevant, as the predecessor can still change the input value. With the change set, we do not know if the value is the same for a logical reason (e.g. because it was saved and restored) or by accident (e.g. because the value 0 got overwritten by the constant 0).

Furthermore, we also maintain the write set in the decode stage, $\text{Wr}_D(E)$. This is used to de-duplicate receive hints (note a receive is a write), and to maintain sequential semantics in case a receive is placed incorrectly (i.e. if it is encountered after writing to a register in the same epoch).

Reattach reached We also track which epochs have *reached* reattach, ReattachReached. Note that the reattach instruction is only committed once the epoch has successfully finished. That is, once it has been committed to architectural state, and reattach is the front of the ROB. Thus, there is a useful state before then, when the epoch has finished all of its instructions, but it cannot yet commit the reattach, as it is not the oldest epoch yet (or it has not finished committing to architectural state). In particular, we can apply the implicit effect of reattach to receive and send all registers (that have not been received and/or sent explicitly). We track this state in $\text{ReattachReached}(E)$.

Correctness and progress The implementation guarantees $\text{InputValue}(E,R)\downarrow$ as soon as there exists an $e < E$ such that

$$R \in \text{Wr}(e) \quad \wedge \quad \forall e'.(e \leq e' < E) \implies (R \in \text{Snd}(e') \vee \text{ReattachReached}(e')). \tag{5.21}$$

That is, if there exists an older epoch e that has written to the register, and all epochs e' between e and E (including e) have either already produced a value in R (if $R \in \text{Wr}(e')$) or it is confirmed that they never will produce it (if $R \notin \text{Wr}(e')$), then the input value of R into E has been set. The youngest e satisfying this condition is the *producer* of R for E.

The above statement can be proven by observing that the right-hand side of the condition becomes true later than the left-hand side (as true writes after the send hint create a conflict), and we call PROPAGATE whenever either of the sub-conditions on the right-hand side change

(and PROPAGATE iterates to all future epochs where the condition is now true). The start of the region requires special treatment (specifically, we set $R \in \mathrm{Wr}(E_0)$ for the starting epoch E_0 for all R for the purposes of this condition, to account for the initial input value being always available.). The full proof is uninteresting but somewhat tedious.

Importantly, however, this property guarantees $\mathrm{InputValue}(E,R)\downarrow$ for all registers R for the oldest active epoch E, as all the epochs e' that are older than E have retired, and thus $\mathrm{ReattachReached}(e')$ is true for all older epochs. Since we define $R \in \mathrm{Wr}(E_0)$ to hold, this implies that eq. (5.21) holds for E, as required. This, in turn, guarantees progress, as the oldest epoch never blocks due to a recv hint.

Note that keeping information about retired epochs is not required at run time, as the pseudo-code does not reference such information.

5.2.6 Partitioning of back-end structures

So far, we have always prioritised older threadlets over younger ones, in order to maximise architectural progress, reduce cross-epoch conflicts and minimise speculative state, as discussed in Section 3.7. In this case, the amount of resources given to each threadlet are only determined by the priorities and which threadlet is ready to take resources. That is, there are no limits on the amount of resources one threadlet can claim.

Back pressure

As discussed earlier (see Section 3.4), when the back-end pipeline cannot match the throughput of the front end, instructions start building up, and the front end is eventually forced to block, due to *back pressure*. Let us consider how back pressure works in the presence of threadlets. Since any threadlet is allowed to utilise any back-end resource, back pressure is exerted by the pipeline at large, and it applies to all threadlets equally. If a structure in the back end is full (regardless of which threadlet created those entries), no threadlet can push any more instructions. In the schemes presented so far in this thesis, this behaviour works relatively well. Due to the prioritisation of older threadlets, if the back end fills up, then it fills (almost) entirely with instructions from the oldest threadlet, unless the oldest threadlet's front end was unable to fill the back end by itself (and spare bandwidth was released to the other threadlets), in which case the pipeline will contain a mix of instructions from different threadlets. There is no fundamental issue in either case, and we can argue that the younger threadlets only got spare bandwidth and otherwise unused resources.

Even though priority inversion is possible due to the lack of preemption, it is only temporary, as each threadlet can either make progress, or it is squashed. However, when we add register value forwarding, this assumption becomes false. The younger threadlet(s) may be waiting to receive register values. If we allow these threadlets to keep dispatching instructions to the back end, eventually we (may) get to a point where the re-order buffer is filled with instructions from threadlets that are blocked, waiting to receive a register value. Those instructions past the receive that do not depend on the received value can execute and write back, but – since commit occurs in order – they still take up slots in the re-order buffer and the load-store queue. If the threadlet that would normally send the values waited on (e.g. the oldest threadlet) cannot get a single allocation, then we have created a circular wait,

and so we suffer a deadlock. Even if there are a few free entries, if the threadlets that are not blocked only get a fraction of the slots normally available in the back end, these threadlets will make progress more slowly. Since the other threadlets are blocked, any cycles wasted here translate directly to additional run time.

Blocking on receive

The simplest solution to this problem is to block as soon as a receive instruction is recognised in the pipeline. This removes the circular waits, and ensures that waiting threadlets hold no resources that they cannot clean up. We can recognise receive instructions in the decode stage (or fetch if the microarchitecture remembers the PC values of all receive instructions), so blocking is possible. However, there are two things that are sub-optimal about this method.

Firstly, some receive instructions will already have the value ready, thus blocking is unnecessary, and it hinders progress. If the value has already been sent, it would be best to just keep going instead. This can be resolved by checking the status of the received register in register rename. We can either block by default in decode (unless we are in the oldest threadlet), and unblock in register rename if the receive instruction already has its value ready, or – alternatively – continue by default, and signal back to decode and fetch from rename that the instructions past the receive should be squashed, and then the threadlet will block.

Secondly, if we block the threadlet upon decoding a receive instruction, once the corresponding register value is received, the threadlet is slow to restart. We can unblock the threadlet instantaneously, but it is likely that most of its in-flight instructions would have committed by the time the value is received, and so the back end pipeline contains very few of this threadlet's instructions, and the front end (past decode) contains none. Thus, it takes valuable cycles before a reasonable out-of-order window can be built up and execution speedup can be restored to its peak (as this requires sufficient ILP from the out-of-order window).

Partitioning space resources

Instead, we can limit the amount of resources allocated to each threadlet. This way, each threadlet is ready to make some progress at any point. More specifically, we can classify resources into bandwidth resources (e.g. issue bandwidth, execution units) and space resources (e.g. ROB entries, IQ entries, LSQ entries). For bandwidth resources, it makes sense to still always prioritise older threadlets, for all the reasons discussed above, but for space resources, we can enforce some partitioning. This way, if plentiful ILP is available in the oldest threadlet, it is still able to use all the bandwidth of the pipeline, but if it has low ILP, then multiple threadlets (normally) have some in-flight instructions ready to be processed at each stage of the pipeline. In particular, this will be the case straight after resuming from waiting on a receive instruction.

The simplest approach would be to statically partition each resource, giving one N'th of it to each of the N threadlets. However, this permanently reduces the allocation of the oldest threadlet by a factor of N, which reduces the amount of available ILP over the whole run time of the system, even when parallelism is limited and even outside parallel regions.

Instead, we can more dynamically split and combine the slices. When T threadlet contexts are *active* (see Section 3.6 for the definition), then we can split the entries into T slices. This way, performance is only impacted when parallelism is actively being exploited.

On the other hand, this dynamic partitioning creates a slight issue when launching a new threadlet. Suddenly, the allocations of the other threadlets drop (as resources are now divided between more threadlet contexts), and there is a chance that some of these older threadlets are now over their new allocations. In the prototype, I handle this by lazily draining out the excess entries. That is, if a threadlet is at or above its maximum allocation for any backend resource, the front end blocks and no new instructions are dispatched until the threadlet is below its allocation limit. Another approach would be to squash excess instructions immediately. The prototype only squashes if a threadlet that is blocked in commit by a receive instruction (i.e. waiting to receive a register value) is over its allocation on a resource that is nearly full between all threadlets. This is necessary, as a waiting threadlet will not commit any instructions until it receives a value, and so it cannot release resources lazily until then, which may cause deadlock if the threadlet that needs to send the register value cannot get entries in the over-allocated resource due to the over-allocation.[8] I measured the performance difference from introducing dynamic partitioning this way (using the optimised version of the scheme presented in Chapter 4), and found no measurable impact beyond noise.[9]

Blocking based on the anticipated (or predicted) wait time for a register value and unblocking shortly before the anticipated receipt of that value would be a more sophisticated approach to this problem, but I opted for the simpler, partitioning-based approach due to a form of selection bias. In particular, we would expect any performance gains from parallelism to be small in regions exhibiting very long wait times (due to insufficient parallelism). Therefore, optimising for this case is not a priority.

5.2.7 Register data flow with iteration packing

There is interaction between iteration packing and register data flow hints. As mentioned in the original description, the base design does not work correctly in the presence of iteration packing. This is because the send hint signifies that the current epoch will not modify the value of the register anymore. However, iteration packing adds the next iteration(s) to the epoch. If the next instance of the body (or header) modifies the register, then we have a problem.

Fixing hint semantics Thus, let us modify the definition of the send hint slightly. Instead of claiming that the rest of the *epoch* will not modify the register, let us just say that the register will not be modified before the next matching reattach instruction. With this new definition, the compiler can insert hints without knowing the packing factor, and the hardware can work out when the final send hint has been encountered, as described below.

[8] In particular, I observed an edge case where the ROB or LSQ were over-allocated by the two older speculative threadlets and fully allocated by the third, all three speculative threadlets were blocked, waiting to receive register values, and the oldest threadlet did not get a single entry as a result, so it could not produce the value.

[9] I observed up to 1% speed difference either way on individual benchmarks, overall yielding a geometric mean speedup of less than 0.1%.

Hardware handling Since the hardware pipeline (specifically the decode stage) tracks which iteration it is currently in (by tracking the iteration packing factor and the number of decoded detach instructions), it can tell which iteration a given send instruction is from. When decoding a send instruction from an iteration other than the last one, we treat it as a no-op, as it does not encode any useful information with respect to the (new, extended) epoch. Send instructions from the last iteration of an epoch are treated as before, as proper hints. Since they signify that the register value is final until the next matching reattach instruction, this means that the register value is final within the epoch, as before (without iteration packing).

Note that receive instructions do not need to be modified. This is because repeated receive hints are already permitted, and handled gracefully. The earliest receive still requires synchronisation, while repeated receives can safely be ignored. This is because the hints do not affect sequential semantics, and so data flow between iterations inside an unrolled epoch are correctly handled, as usual. Correct cross-epoch data flow only requires that there is a receive instruction before the first read of the register, but it is not mandated which iteration that receive comes from.

Residual limitations While this restores correctness, note that performance is still far from ideal in loops with high iteration packing factors. Since the send in the last iteration in epoch E needs to synchronise with the first receive in epoch $E+1$, the only remaining fully-parallelisable code is anything that comes after the last send in E and anything that comes before the first receive in $E+1$. Assuming iterations are of a similar size, both send and receive are executed in each iteration, and send is after receive, these regions combined only make up at most one iteration's worth of instructions per epoch, or $\frac{1}{P}$ of each epoch under a packing factor of P. Even if we assume these fully parallel parts to completely overlap with other work (and thus amortise away), the maximum speedup from this is limited by $\frac{P}{P-1}$, which gets very small for $P >> 1$. Even though we can also partially parallelise the region between the first receive and last send with other similar regions (i.e. instructions that are independent of the receive can go early, as long as they are fetched into the threadlet's out-of-order window before the receive is committed), speedups may still suffer significantly in many cases due to these limitations.

This issue could be solved or improved if the compiler performed some degree of unrolling (instead of, or on top of, iteration packing), in order to reduce the packing factor P, and thus increase the speedup potential. In this case, the placement of send instructions needs to consider unrolling for correctness. In particular, the compiler should only insert a send hint after the last modification in the bigger, unrolled iteration.

To increase the size of the fully parallelisable parts of the iteration, the distance between the first receive and send hints has to be reduced. After unrolling, this may be possible using code motion optimisations in the compiler (e.g. moving all updates to the register with the loop-carried dependence close together). These optimisations are left to future work in compiler research.

Value prediction

One logical extension to the microarchitectural implementation is to predict values we are waiting to receive. It may be possible that a predictable pattern is not obvious at compile time, or it may be the case that a variable updated in the middle of the iteration is typically updated following a predictable (e.g. strided) pattern, but diverges from this in some iterations, or that conflicts are rare at run time, but this is difficult to derive statically. In these cases, it is challenging for the compiler to move the updates into the header or continuation, or otherwise derive the variable (e.g. compute it from an induction variable).

Value prediction here is more well-bounded, and likely more impactful than it is for long-latency loads and other cases where we might apply it in microarchitecture, thus the trade-offs for applying it may be worthwhile.

The predictors used can be arbitrarily complex, but simple predictors (e.g. constant or stride) predictors may eliminate cases such as rare dependences, unrecognised induction variables and mostly-strided variables. Investigation of the frequencies of these cases, the cost of prediction, and the best predictors to choose is left to future work. The prototype does not implement value prediction.

5.2.8 Case study and evaluation

Since compiler-based insertion of hints is left to future work, to test the implementation, I investigated loops in the SPEC CPU2006 suite by hand. I used our profiling tool, OptiWISE [27], to help with this process. OptiWISE generates high-level statistics about loops and control flow, as well as giving a clear view of how expensive each instruction is by combining execution count and time-based sampling information in a real system.

Due to the limitations with respect to iteration packing (described above), I restricted my attention to loops that were at least 200 instructions per average iteration. Further to this, I ignored loops that accounted for less than 5% of workload execution time, and those that would likely not be profitable based on other characteristics[10] to reduce the amount of manual work. Discounting loops with incorrect line information[11], I obtained 27 candidate loops.

Out of these, 9 were already covered well without register data flow hints, 8 only contained false dependences in the compiler's intermediate representation[12], which can be eliminated with further development work, 2 contained floating-point reductions that can be handled using the reduction engine. Out of the remaining 8 loops, 5 exhibited complicated or long dependence chains. For some of these 5 loops, value prediction may help, and at least two of them were simply too complicated for full manual inspection. The remaining three loops contained cross-iteration scalar dependences in LLVM intermediate representation as the main bottleneck.

[10] Specifically, I discarded loops with more than 100,000 average dynamic instructions per iteration, fewer than 2,000 average dynamic instructions per invocation, or 4 or fewer average iterations per invocation.

[11] Unfortunately, this included the full Fortran subset due to issues with the flang front end.

[12] This included write-after-write dependences, which show up as read-after-writes due to interjecting phi nodes and global variables used locally, which had to be reloaded each iteration only due to the lack of accurate alias analysis.

For one of the three loops, the dependences were lowered to memory in the back end compiler, due to register pressure. Manual code changes to try to prevent this were unsuccessful, making this loop unsuitable.

I successfully annotated the remaining two loops – one in *sphinx3* and one in *bzip2* – by hand, and achieved speedups, as detailed below.

Annotation methodology

I performed the annotation by defining a handful of C preprocessor macros in a header file, one for each hint (i.e. `detach`, `reattach`, `sync`, `send` and `recv`), which insert the hints using inline assembly (using the *volatile* qualifier to signal side-effects). I defined a unique label at the continuation point, which was referenced by each hint to encode the region address, and the register argument of send and receive was piped in via the standard inline-assembly syntax (restricting the allocation of the variable to a register). I added the new hints to LLVM's TableGen, to define their machine encodings. This was sufficient to let the new hints flow through LLVM and output the intended, correct machine code for these loops.

In addition to annotating the hints, I manually encoded a couple of reduction updates into the correct format – recognised by the microarchitecture – using inline assembly. This was straightforward and can easily be automated in the compiler. Furthermore, I also performed a handful of simple code motion changes to eliminate dependences and reduce the distance between a recv hint and its corresponding send.

Example: sphinx3

The example loop in *sphinx3* covers 46% of the execution time of the benchmark, and an extract of it is shown in Listing 5.6. The loop contains two variables (called `pbest` and `best`), which calculate the maximum among elements seen so far. One (`best`) is a pure reduction (but cannot be handled by the current prototype, as the maximum reduction operator is not implemented), while the other (`pbest`) is used to compute a value, which is used in a condition check. I surrounded the uses of these two variables with register data flow hints (as shown in Listing 5.6), annotating once along each of three possible conditional paths through the loop. Further to these, the loop contains two scalar reductions (summations, `ng` and `ns`), which I lowered to memory. It was necessary to manually encode the increments in assembly, as LLVM lowered them differently by default. No other manual code modifications were necessary.

Running the whole benchmark (through the whole SimPoint flow, but with no loop selection) with this one loop annotated yields a 9.8% speedup, which translates to a 27% in-region speedup.

Example: bzip2

This benchmark contains a more complex example loop, with branching and two inner loops. There are two scalar variables (called `zPend` and `wr`) and two arrays (named `yy` and `mtfFreq`) that contain loop-carried dependences. The loop updates `zPend` conditionally based on the

Listing 5.6: Example usage for the register data flow extension (extract from `sphinx3`). Annotations are shown in . Frequent cross-iteration dependences on the variables `pbest` and `best` are handled using the annotations `RECV` and `SEND`. Dependences on `ng` and `ns` are handled as reductions (in memory).

```
1  for (s = 0; s < /*...*/; s++) {
2      if (/*...*/) {
3          /*...*/
4          RECV(pbest, best);
5          if (pbest < senscr[s]) pbest = senscr[s];
6          if (best < senscr[s]) best = senscr[s];
7          SEND(pbest, best);
8          ng += mgau_n_comp(g,s);
9          ns++;
10     } else {
11         if (/*...*/) {
12             RECV(pbest); SEND(pbest);
13
14             if(pbest < /*...*/){
15                 ng += approx_mgau_eval (/*...*/, senscr);
16                 ns++;
17             } else { /*...*/ }
18
19             RECV(best);
20             if (best < senscr[s]) best = senscr[s];
21             SEND(best);
22         } else { RECV(pbest, best); SEND(pbest, best); }
23     }
24     /*...*/
25 }
```

value of `yy[0]`: either `zPend` is simply increased, or the first inner loop is repeated until `zPend` is reduced to 0 using right-shifts. Thus, the first loop is only executed $\log_2(\texttt{zPend})$ times, where `zPend` is the number of outer loop iterations since the last invocation of the inner loop, and so this loop is relatively cheap. I annotated `zPend` with send and receive hints.

The other variable, `wr` is updated using increments, and it tracks elements of an output array that have been filled. Naïvely annotating its increments with hints will put the send hint at the very end of the loop recv near the start. However, the only late update to `wr` is a single, unconditional increment, thus I hoisted this increment up as far as possible, and replaced later usages of `wr` with $\texttt{wr} - 1$. Inserting data-flow hints this way exposed some parallelism.

In addition to the scalar dependences, I encoded the reductive (increment) updates to an array (`mtfFreq`) in the reduction format recognised by the compiler, but the array `yy` still yielded frequent conflicts. Closer inspection of the code revealed that the second inner loop searches for a specific value in `yy`, swapping it to the front (by shifting all previous elements back by one space). The value searched for is known at the start, but the original algorithm only writes this back to `yy[0]` after exiting the inner loop. It was easy to manually modify the code to perform this write earlier, and I believe the independence of all pointers can be derived using scalar evolution analysis in a compiler, although automating the detection of such dependences (and detecting when they matter) may not be straightforward in a compiler.

With all the modifications included, I managed to obtain speedups for this benchmark. The extent of the speedups depended on the input file, ranging from 6.2% for the input called *program* to 0.6% for *source*. One input set (*text*) observed a 0.5% slowdown due to very short loop iterations (fewer than 25 instructions per iteration on average). Overall, this loop (covering 97% of the benchmark's run time) achieved a modest 2.4% speedup.

5.3 Frequent true through-memory dependences

If through-memory dependences are infrequent enough, we already support them efficiently through trial and error, using conflict checking and squashing. For frequent dependences, this method is not effective, as it leads to continuous and repeated squashing, and a lack of progress, as detailed below. There are multiple viable paths to handling frequent through-memory dependences, which I briefly discuss here for completeness. These have not been implemented in the prototype, and they require more involved compiler support.

5.3.1 Problem

There are two closely related issues when applying our current methods to loops with frequent true through-memory dependences: late and repeated squashing.

Late squashing

If a conflict between epoch E and epoch $E+1$ occurs late enough, then E is already close to finishing its execution, and thus squashing and restarting $E+1$ leads to little or no additional parallelism being exposed. In fact, a slowdown may be observed, in the case where the

threadlet of E has already fetched the final reattach instruction. Furthermore, if all epochs exhibit such conflicts, then parallelising the loop becomes overall impossible this way, due to the dependence.

Note that we already support data forwarding, and we eliminated all but true dependences. Thus, this case arises only if E writes to a memory location M towards the end of its lifetime, and $E+1$ already loaded a value from M (earlier in its lifetime). Therefore, the conflict is very much a property of the code, and the timings of how the code is executed.

Repeated squashing

A similar, but slightly different issue is when epochs get repeatedly squashed. For example, epoch $E+1$ starts by reading a memory location M, and epoch E writes to this location many times during its lifetime (e.g. inside an inner loop). In this case, currently, we restart $E+1$ many times. In some pathological tests, I observed up to hundreds of restarts per epoch. This is obviously not ideal. If combined with late squashing (e.g. if E keeps writing M until the end of its life), then the problem is the same as above (late squashing), but repeated squashing can be a problem even if it does not lead to particularly late restarts.

For example, epoch E may start with an inner loop that writes M (e.g. popping the front element from a data structure), but then continue to do significant work that is typically independent from M. In this case, epochs E and $E+1$ could be (partially) overlapped, and provide a benefit. However, the repeated restarting of $E+1$ resulted in resources being unnecessarily held by $E+1$ (from E and older epochs). Due to the frequent restarts, it is unlikely that $E+1$ would have made any significant progress, and so it likely did not do any useful prefetching work either. Thus, the resources used by it were wasted, resulting in a slowdown.

5.3.2 Possible solutions

The cleanest solution is to make the compiler move frequent dependences (and their dependence chains) into the header or continuation, however this requires complex compiler analysis and transformations, and it may highly reduce the remaining body size. In particular, if we move the set of instructions S, then the compiler needs to ensure that any instructions S_{gen} that are involved with generating addresses for S and any instructions $S_{conflict}$ that *may* conflict with S are also moved, i.e. $S_{gen} \cup S_{conflict} \subseteq S$, and if it is moving code to the continuation, it also needs to move any dependent instructions S_{dep} too (i.e. $S_{dep} \subseteq S$), unless code can be cleverly (and profitably) duplicated to prevent other code from having to be moved. For example, if a set of dependent instructions S_{dep} only depend on a single instruction $I \in S$, and we are moving S to the continuation, then we can instead copy I to make I', and keep I' and S_{dep} in the body (making S_{dep} read the result of I'), and still move S.

Another way of cutting down on the necessary movement is additional synchronisation hints (such as threadlet barriers), which could be used to create sequential regions. A simple design proposed by prior work [31] introduces a barrier that can only be passed once the current threadlet has become architectural, thus allowing for a sequential region at the end of each epoch. Another possibility is to store data destined for the next epoch into memory, and use register data-flow hints to pass the address to the next epoch (thus also signalling

that the data is ready). The compiler impact, and interplay with iteration packing, both raise concerns, and exploring these concerns is left to future work.

A possible fully microarchitectural approach to handling repeated (but not late) squashing may be to monitor frequently conflicting addresses (or program counters), and delay reads accordingly in order to eliminate hazards. The complexity here is to accurately predict the amount of delay needed. This could be a fixed time delay, waiting for a set number of accesses, or waiting until the predecessor threadlets have passed a certain point (e.g. an often-conflicting store instruction). While such monitoring and prediction may lead to speedups, the complexity is high, and we can expect results to be highly sensitive to design details. Therefore, I leave the exploration of this to future work.

Chapter 6

Conclusions

In the previous chapters I have argued that in-core, hint-based, speculative multithreading is a promising technique to combine the benefits of instruction-level and thread-level parallelism to exploit medium-grained parallelism (Chapter 1), which is not normally exposed by traditional out-of-order execution and multithreading. I have shown that the initial argument that this can be added with minimal disruption is correct (Chapter 3) by giving a working architecture (co-developed with others) and microarchitectural design (developed by myself), and describing a working compiler (developed by others). Then, I improved performance, using a series of optimisations (Chapter 4), and showed meaningful performance improvements. Finally, I described some ways for handling dependences, focusing on specific types, in hardware or in the architecture (Chapter 5). This chapter briefly summarises the limitations and future work, before re-visiting the hypothesis shown in Chapter 1, and presenting an outlook for applying these techniques to future microarchitectures.

6.1 Limitations and future work

Table 6.1 summarises future work by high-level area. It also makes references to which section of the thesis discusses each issue. I give a high-level summary here, but please refer back to the original sections for context.

Parallel patterns There are parallel patterns that are not currently captured (well) by the prototype. These include nested loops, where we only parallelise on a single level (and do not currently support recursive nesting), non-loop quasi-independent regions (such as function calls and their continuations), and loops requiring more relaxed placement rules for the hints, such as those that only find parallel work on some iterations (e.g. detach and reattach could sensibly go on one conditional path only). Selecting the best regions is currently achieved by simulating perfect static loop selection using filtering during our experiments. Static loop selection (in the compiler, perhaps using profile-guided optimisation), and dynamic profiling of performance (for loop selection and other microarchitectural optimisations) are also left to future work.

General area	Area of future work	Discussed in
Parallel patterns	Nested parallelism	Section 3.2.5
	Non-loop continuations	Section 3.2.6
	Only some iterations detach	Section 3.3
	Loop selection and run-time profiling	Section 3.12
Detailed implementation	Instruction fetch	Section 3.8.2
	SSB organisation	Section 3.10.2
	Epoch commit mechanism	Section 3.10.6
	Conflict checker circuit	Section 3.11.2
	Optimal resource allocation	Section 5.2.6
Dependences	Compiler analyses and transformations	Chapter 5
	Infrequent register dependences	Section 5.2.2
	Frequent memory dependences	Section 5.3

Table 6.1: Limitations and future work, with reference to where they were discussed.

Detailed implementation Although I attempted to keep the design of the scheme feasible to implement by using a reasonable organisation and mapping to known concepts or sensible logic, I do not provide a detailed, circuit-level (register-transfer level, RTL) implementation. This is due to the fact that such an implementation will inevitably take significant engineering effort. To be convincing, it should rely on a state-of-the-art design, and go through thorough optimisation. Nevertheless, there will no doubt be interesting challenges arising here. The areas listed in table 6.1 are of particular interest due to possible foreseen challenges. Instruction fetch, which could not be modelled accurately in gem5 due to its model frontend differing significantly from cutting-edge designs. For several structures, such as the branch predictor and prefetcher, I used an off-the-shelf, speculation-unaware version. Although our analysis suggests these worked well and did not cause obvious pathological patterns, future work may find scope for improvements here, as the evidence is inconclusive due to noise.[1] The speculative state buffer's internal organisation (e.g. banks, pipeline, exact timings and placement) was treated as a black box, and since this is a novel part of our design, future work should produce a full RTL design for this component, and evaluate area and power accurately. Without RTL, these numbers are impossible to estimate accurately. In the thesis, I used conservative estimates to account for the lack of a detailed implementation, but these can no doubt be improved. Furthermore, the way in which epochs are atomically committed has been logically described in detail, but such additions to the coherence protocol are tricky, as evidenced by the well-known challenges [89] in commercially deploying hardware transactional memory. Similarly, the conflict checking logic has been described in detail, but due to it being a novel component, future work should provide a full RTL design to find and evaluate challenges, and accurately estimate area and power overheads. Finally, the partitioning of resources is currently done in a relatively simple way (i.e. dual

[1] For example, a small observed performance improvement may be the result of a large speedup lost due to high overheads, or it could just be a small speedup with low overheads. Similarly, cache miss rates can and do go up and down significantly due to the increased run-ahead and prefetching effects. For example, failed speculation may increase miss rates if bogus data leads down an incorrect execution path, but it may also reduce miss rates if the same loads are replayed with the same input addresses.

scheduling, as described in Section 5.2.6), but in theory this could be improved by estimating the probability of success from extending each given threadlet's window, and allocating resources accordingly. Exploring this is left to future work.

Dependences Throughout the thesis, and specifically in Chapter 5, I repeatedly mention places where the compiler could do more. Specifically, for dependence handling, identifying conflicting operations and either annotating them (via some additional hints), or performing transformations that increase the non-conflicting overlap between epochs, altogether eliminating a conflict, or turning it into a special pattern that can be handled (e.g. reduction) are promising areas. Furthermore, handling the two types of dependences shown in table 6.1 effectively is left to future work. Tackling these patterns could involve microarchitectural techniques, compiler techniques, or a software-hardware co-design, and it could unlock further speedups beyond those shown in this thesis.

6.2 Hypothesis revisited

> **Hypothesis:** *In-core, hint-based, speculative multithreading can be added to a realistic high-performance out-of-order superscalar processor pipeline, with minimal impact on the instruction set architecture, and without disturbing existing microarchitectural techniques and optimisations. Thus, adding this new capability produces meaningful, practical performance gains over a state-of-the-art baseline.*

In the introduction, I set out to test the above hypothesis. Having discussed the motivation, related work, base technique, optimisations and remaining challenges, I now summarise the findings in this thesis, and argue that they support this hypothesis.

In Chapter 3, I presented a design that shows how to add in-core, hint-based, speculative multithreading to a realistic high-performance processor. I presented a hint-based architectural extension that does not affect sequential semantics, thus it has a minimal impact on the instruction set architecture and other architectural components (e.g. the coherence model, inter-core timings). I also demonstrated how hardware can interpret and take advantage of these hints, in order to expose additional medium-grained parallelism. Doing this preserved compatibility with existing cutting-edge microarchitectural techniques and optimisations, and should not result in significant sequential slowdowns (due to the transparent design). This chapter showed the feasibility of the technique and achieved a measurable 6.4% geometric mean speedup on the SPEC CPU2006 benchmark suite.

In Chapter 4, I identified bottlenecks, and added a set of optimisations to the system. Without breaking compatibility or impacting sequential performance, these optimisations allowed us to improve parallel speedups, increasing to 11.8% geometric mean for CPU2006. The dynamic instruction coverage (time on) was identified as a limiting factor, largely due to cross-iteration dependences.

Finally, in Chapter 5, I investigated dependences in more detail. I identified two classes of dependences – reductions and similar associative update patterns (Section 5.1) and frequent register dependences (Section 5.2) – that are more easily handled. I presented a

fully microarchitectural solution to eliminate conflicts on in-memory reductions and other associative update patterns. I also described a novel hint-based architectural extension that can encode frequent register dependences, I presented a microarchitectural implementation of this extension, and I evaluated it on hand-crafted examples from two benchmarks in the SPEC CPU2006 suite. I also briefly discussed the challenges and potential solutions for other dependences.

In conclusion, I showed the implementability of in-core, hint-based, speculative multi-threading, demonstrated compatibility with existing, cutting-edge microarchitectural techniques and modern instruction set architectures, and I demonstrated that meaningful performance gains (11.8% geometric mean, up to 3.3× maximum per loop, and 1.74× per benchmark) can be achieved adding it to a realistic, high-performance, out-of-order processor. This proves each point mentioned in the hypothesis.

As discussed in Section 6.1, some limitations remain, and certain areas were left unexplored. Exploring these is left to future work, opening a relatively wide space for further research.

6.3 Future outlook

Before wrapping up this thesis, let us discuss how the relevance and usefulness of the techniques presented may change in the future. As discussed in Chapter 1, we expect benefits from capturing parallelism that is not captured by other existing techniques. Clearly, with microarchitectural changes and improvements, the parallelism that is captured by different techniques changes, which changes the primary targets for threadlet-based speculation. Although it is impossible to accurately predict future improvements, making an educated guess is a necessary part of the research-development cycle, due to the timelines involved in research, development and productisation, and the pace of improvements in industry. I present my best guess in this section, based on current trends.

Overall, I argue that the primary target loops (as considered in Figure 1.4) will likely shift towards bigger and more complex loops, as shown in Figure 6.1, but the relative impact of these loops on CPU performance may also grow. This would make the techniques presented in this thesis more relevant, but also more challenging, with future work required specifically on better dependence handling and code generation. I give some details below.

ILP improvements With wider and deeper processor pipelines and better branch prediction and prefetching (continuing recent trends [21, 30, 67, 71]), we can expect high-performance CPUs of the future to be able to find and exploit more instruction-level parallelism. This would manifest in two ways. Firstly, it would allow processors to find ILP between instructions that are further apart in the dynamic instruction stream, thus more effectively finding parallelism between iterations in loops with more complex and longer iterations. This would effectively push the 'bottom' boundary in Figure 6.1 up. Secondly, it would increase performance (specifically, IPC) in high-ILP regions that currently saturate pipeline resources and have additional, unexploited ILP left. This would make under-utilisation greater in any regions where the available ILP is already (nearly) fully exploited, and it also increases the importance of performance in these under-performing regions, due to Amdahl's law [2] (i.e.

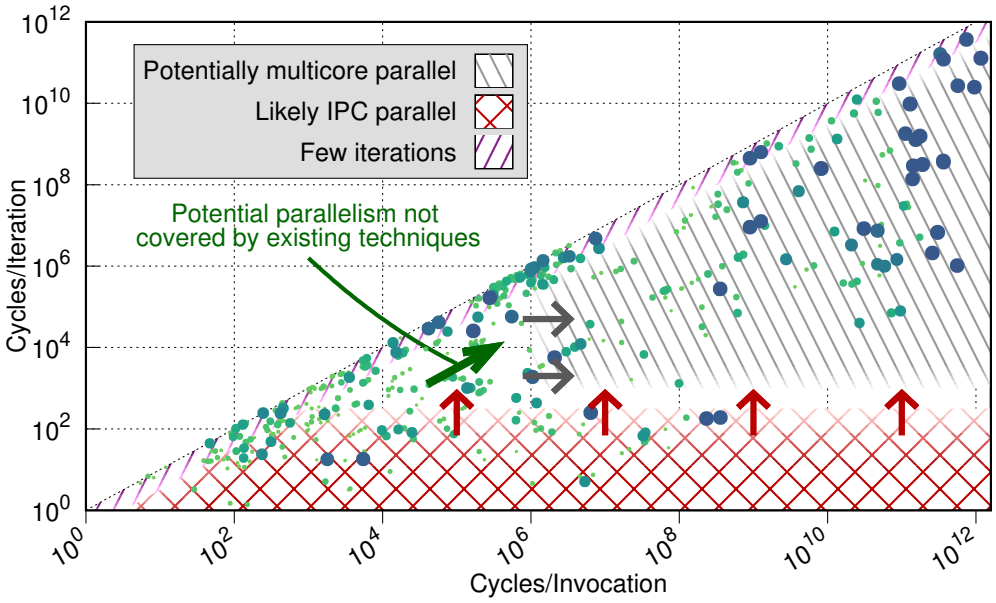


Figure 6.1: Possible future change in target loops. ILP may cover more and larger loops, but multicore parallelism will likely remain difficult, and the overheads may grow in relative terms. Adapted from Figure 1.4.

as the rest of the code finishes faster, slow regions contribute more to overall execution time in relative terms).

TLP changes With larger, more complex cores, we can expect cross-core collaboration (multithreading) overhead to be higher. This is due to both absolute and relative increases in costs. In absolute terms, larger cores and larger caches lead to higher access latencies, and the impact of contention also becomes higher. In relative terms, due to the cores being faster individually, the fixed costs will count for more. For example, a 100-cycle delay takes as long as executing 400 instructions when IPC is 4, but as long as 800 instructions when IPC is 8. Thus the length of a parallel section (in instructions) where costs amortise away can be expected to grow with larger, more powerful cores. This pushes the 'right-hand side' boundary in Figure 6.1 towards the right. Additionally, the explicit parallelisation of (more complex) loops will continue to be challenging (unless fundamental shifts occur in the programming model). And finally, even if thread-level parallelism increases performance, it is likely that these large cores will – individually – still be under-utilised (unless significant ILP also exists), thus leaving space for threadlet-based speculation on top.

Accelerators With 'emerging' workloads (such as machine-learning training and inference) having become mainstream already, and the research community gaining a better understanding of them, it is likely that the set of custom accelerators and alternative architectures (e.g. GPUs, TPUs), and their performance will continue to grow. Workloads that are standard and particularly amenable to easy parallelism are far easier to map to these substrates, therefore we can expect that code destined for the CPU will encode the remaining, harder-to-parallelise work. This would have two implications. Firstly, sequential performance, and performance

on code that cannot easily be parallelised, may become more important for CPUs. Secondly, it means that any threadlet-based speculation scheme will need to handle more complex loops.

Summary Thus, it is reasonable to expect that the space of unexploited parallelism will not disappear in the future, but we can expect target loops to become larger (on average), and more complex. Thus, future work needs to concentrate on tackling these workloads. It is worth noting that the increased state required to handle large iterations (with larger working sets) is also more feasible to add to a larger core (as the relative overhead is smaller for the same area). Future work may experiment with different data organisation, for example via banked or multi-level speculative state buffers.

Appendix A

Glossary of terms

Term or acronym	Definition
Speculative multi-threading (SpMT)	Attempt to exploit thread-level parallelism speculatively. Typically, speculation is used to cut unknown or uncertain cross-thread dependences, to predict future program points to launch threads, or to predict (some) input values.
Thread-level speculation (TLS)	Synonym for *SpMT*.
Instruction-level parallelism (ILP)	Very fine-grained parallelism exploited between nearby dynamic instructions, typically by an out-of-order processor pipeline.
Data-level parallelism (DLP)	Regular parallelism, typically exploited by single instruction multiple data (SIMD) processors or short-vector instruction set extensions.
Thread-level parallelism (TLP)	Coarse-grained parallelism between multiple threads.
Simultaneous multi-threading (SMT)	Supporting multiple hardware thread contexts within each processor core, each running at the same time and dynamically sharing core resources.
Chip multi-processor (CMP)	A chip with multiple processor cores.
Architectural state	The program counter, architectural registers, memory contents and OS state of the process. The program's architectural state is always well-defined.
Threadlet state	The execution state of the threadlet. Once the threadlet is *architectural*, this is the *architectural state* of the program, otherwise the threadlet state is speculative, consisting of the contents of the threadlet's architectural register file, program counter and buffered memory updates.
Speculatively committed	An instruction is speculatively committed if it has passed all stages of the pipeline, irrespective of the state of its threadlet. A memory write is speculative committed if it has reached the SSB.
Architecturally committed	An instruction whose effects have been applied to the *architectural state*. A *speculatively committed* instruction is *architecturally committed* once/if its associated threadlet becomes architectural.
Thread	An architecturally visible unit of execution with its own program state (program counter, architectural registers) and process information. Controlled by the operating system.
Threadlet	A unit of execution with its own program counter and architectural registers (like a thread), but not exposed architecturally. Threadlets are controlled by the microarchitecture, and execute parts of the same program.
Threadlet context	The hardware resources within a processor core capable of running and managing a threadlet.
Parallel region	A program region within which speculative parallelisation takes place. Different regions are disjoint, and each has a unique parallelisation strategy associated with it.

Epoch	A contiguous slice of the dynamic instruction stream that forms the unit for parallelisation. Different epochs may be parallelised with each other, whilst ILP is exploited within each epoch. Each epoch is mapped to a threadlet.
Architectural threadlet	A threadlet whose effects have been applied to architectural state. Only one active threadlet can be architectural at any given time.
Speculative threadlet	A threadlet that is not architectural.
Static instruction	An instruction in the program binary. The same static instruction may be executed multiple times during the lifetime of the program.
Dynamic instruction	An instruction in the dynamic instruction stream at run time.
Program order	The architectural ordering of dynamic instructions in the instruction stream at run time.
Sequential semantics	The observable behaviour of the program as defined by the instruction set. This includes the results of a single-threaded computation, as well as the ordering of externally-visible events (including consistency).
Consistency	Rules in the instruction set for reordering memory operations. With respect to a dynamic instruction stream, consistency defines a partial order of memory operations.
Coherence	The property of the memory system that there may only be one well-defined value (the 'architectural value') for each memory location within main memory and all coherent caches. Non-coherent buffers and caches may contain other copies, but these only become observable globally once written back to the coherent memory system.
Memory model	The combination of consistency and coherence rules set out in the instruction set architecture.
Fork-join parallelism	A paradigm dividing the program into alternating sequential and parallel segments to expose *thread-level parallelism*.
Asymmetric fork-join	An extension of the *fork-join parallelism* paradigm, where *program order* remains well-defined (even between parallelised segments).
Data flow	A true data dependence. Data flow occurs between an instruction that produces output data, and the instructions that consume that data. Data flow occurs between a pair of program regions if it occurs between any pair of instructions in those regions (during any execution of those region).
Write-after-Write (WaW)	A pair of (typically overlapping) write instructions. Program order defines the correct final value of the targeted memory (if overlapping), or the order in which the operations must be exposed to other cores.
Read-after-Write (RaW)	A write and an overlapping read later in program order, such that the read should observe the effects of the write.
Write-after-Read (WaR)	A read and an overlapping write later in program order. The read should never observe the effects of the write.
Dependence, hazard	One of the above three patterns (WaW, RaW, WaR) with the two operations split between different epochs. Naïve parallelisation may reorder the two operations and thus may violate sequential semantics.
Conflict	A dependence (hazard) that has been violated by parallelisation. The system must take corrective action that both prevents the conflict from being observed externally (i.e. preserve consistency) and from changing single-threaded behaviour.

Bibliography

[1] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, J. O. Bondi and J. Smith, Eds., 1998. Available: https://doi.org/10.1109/MICRO.1998.742784

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the American Federation of Information Processing Societies (AFIPS)*, 1967. Available: https://doi.org/10.1145/1465482.1465560

[3] M. Arif and H. Vandierendonck, "Reducing the burden of parallel loop schedulers for many-core processors," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018. Available: https://doi.org/10.1145/3178487.3178517

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint*, 2015. Available: https://arxiv.org/abs/1508.03619

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011. Available: https://doi.org/10.1145/2024716.2024718

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. Available: https://doi.org/10.1145/362686.362692

[7] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the Design Automation Conference (DAC)*, 2007. Available: https://doi.org/10.1145/1278480.1278667

[8] G. Bronevetsky, J. Gyllenhaal, and B. R. De Supinski, "CLOMP: accurately characterizing OpenMP application overheads," in *Proceedings of the International Workshop on OpenMP (IWOMP)*, 2008. Available: https://doi.org/10.1007/978-3-540-79561-2_2

[9] J. Burns and J.-L. Gaudiot, "Smt layout overhead and scalability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, 2002. Available: https://doi.org/10.1109/71.983942

[10] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: an architecture-compiler co-design for automatic parallelization of irregular programs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014. Available: https://doi.org/10.1109/ISCA.2014.6853215

[11] S. Campanoni, T. M. Jones, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix: making the extraction of thread-level parallelism mainstream," *IEEE MICRO*, vol. 32, no. 4, 2012. Available: https://doi.org/10.1109/MM.2012.50

[12] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's Rock processor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009. Available: https://doi.org/10.1145/1555754.1555814

[13] N. C. Crago and S. J. Patel, "Outrider: efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 117–128. Available: https://doi.org/10.1145/2000064.2000079

[14] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6. Available: https://doi.org/10.1109/HPEC.2013.6670342

[15] L. Developers, "LLVM documentation: vectorization plan." Available: https://llvm.org/docs/VectorizationPlan.html

[16] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011. Available: https://doi.org/10.1145/2000064.2000108

[17] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM Computing Surveys*, vol. 49, no. 2, 2016. Available: https://doi.org/10.1145/2938369

[18] S. Eyerman, W. Heirman, S. Van Den Steen, and I. Hur, "Enabling branch-mispredict level parallelism by selectively flushing instructions," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021. Available: https://doi.org/10.1145/3466752.3480045

[19] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic job scheduling on the IBM POWER8," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2016. Available: https://doi.org/10.1109/HPCA.2016.7446103

[20] M. Franklin, “The multiscalar architecture,” Ph.D. dissertation, School of Computer, Data & Information Sciences, University of Wisconsin at Madison, USA, 1993, uMI Order No. GAX94-07362.

[21] A. Frumusanu, “Apple announces the Apple silicon M1: ditching x86 - what to expect, based on A14,” 2020. Available: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

[22] P. Ginsbach and M. F. P. O’Boyle, “Discovery and exploitation of general reductions: a constraint based approach,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2017. Available: https://doi.org/10.1109/CGO.2017.7863746

[23] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, p. 5–48, 1991. Available: https://doi.org/10.1145/103162.103163

[24] S. C. Goldstein, K. E. Schauser, and D. E. Culler, “Lazy threads: implementing a fast parallel call,” *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 5–20, 1996. Available: https://doi.org/10.1006/jpdc.1996.0104

[25] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, “Speculative versioning cache,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 1998, pp. 195–205. Available: https://doi.org/10.1109/HPCA.1998.650559

[26] J. Gray *et al.*, “The transaction concept: Virtues and limitations,” in *VLDB*, vol. 81, 1981, pp. 144–154.

[27] Y. Guo, A. W. Chadwick, M. Erdos, U. Bora, I. Vougioukas, G. Gabrielli, and T. M. Jones, “OptiWISE: combining sampling and instrumentation for granular CPI analysis,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2024. Available: https://doi.org/10.1109/CGO57630.2024.10444771

[28] T. J. Ham, J. L. Aragón, and M. Martonosi, “Desc: decoupled supply-compute communication management for heterogeneous architectures,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 191–203. Available: https://doi.org/10.1145/2830772.2830800

[29] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “SimPoint 3.0: faster and more flexible program phase analysis,” *Journal of Instruction Level Parallelism*, vol. 7, no. 4, 2005. Available: http://www.jilp.org/vol7/v7paper14.pdf

[30] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D’Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, “Haswell: the fourth-generation Intel Core processor,” *IEEE Micro*, vol. 34, no. 2, 2014. Available: https://doi.org/10.1109/MM.2014.10

[31] L. Han, W. Liu, and J. M. Tuck, "Speculative parallelization of partial reduction variables," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2010. Available: https://doi.org/10.1145/1772954.1772975

[32] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

[33] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006. Available: https://doi.org/10.1145/1186736.1186737

[34] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: Association for Computing Machinery, 1993, p. 289–300. Available: https://doi.org/10.1145/165123.165164

[35] N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. C. Pocock, G. Brown, M. Luján, I. Watson, and M. Cintra, "Toward a more accurate understanding of the limits of the TLS execution paradigm," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010. Available: https://doi.org/10.1109/IISWC.2010.5649169

[36] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: an industry perspective," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021. Available: https://doi.org/10.1109/ISPASS51385.2021.00034

[37] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015. Available: https://doi.org/10.1145/2830772.2830777

[38] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465. Available: https://doi.org/10.1109/ISCA.2018.00045

[39] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "Tight analysis of the performance potential of thread speculation using spec CPU 2006," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2007. Available: https://doi.org/10.1145/1229428.1229475

[40] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "On the performance potential of different types of speculative thread-level parallelism," in *Proceedings of the Annual International Conference on Supercomputing (ICS)*, 2006. Available: https://doi.org/10.1145/1183401.1183407

[41] T. Knight, “An architecture for mostly functional languages,” in *Proceedings of the ACM Conference on LISP and Functional Programming*, 1986. Available: https://dl.acm.org/doi/10.5555/106626.106648

[42] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: exploiting speculative execution,” in *Symposium on Security and Privacy (S&P)*, 2019. Available: https://doi.org/10.1109/SP.2019.00002

[43] V. Krishnan and J. Torrellas, “Executing sequential binaries on a clustered multithreaded architecture with speculation support,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 1998.

[44] M. S. Lam and R. P. Wilson, “Limits of control flow on parallelism,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1992. Available: https://doi.org/10.1145/139669.139702

[45] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis and transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004. Available: https://doi.org/10.1109/CGO.2004.1281665

[46] Y. Li, K. Skadron, D. Brooks, and Z. Hu, “Performance, energy, and thermal considerations for SMT and CMP architectures,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2005. Available: https://doi.org/10.1109/HPCA.2005.25

[47] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996. Available: https://doi.org/10.1145/237090.237173

[48] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, “The gem5 simulator: version 20.0+,” *CoRR*, vol. abs/2007.03152, 2020. Available: https://arxiv.org/abs/2007.03152

[49] J. Mak and A. Mycroft, "Limits of parallelism using dynamic dependency graphs," in *Proceedings of the International Workshop on Dynamic Analysis (WODA)*, 2009. Available: https://doi.org/10.1145/2134243.2134253

[50] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *Proceedings of the International Conference on Supercomputing (ICS)*, G. K. Egan, R. P. Brent, and D. Gannon, Eds., 1998. Available: https://doi.org/10.1145/277830.277850

[51] T. McMichen, N. Greiner, P. Zhong, F. Sossai, A. Patel, and S. Campanoni, "Representing data collections in an SSA form," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2024. Available: https://doi.org/10.1109/CGO57630.2024.10444817

[52] M. Mitzenmacher and E. Upfal, *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge University Press, 2017.

[53] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[54] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, 2009. Available: https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf

[55] Q. M. Nguyen and D. Sanchez, "Pipette: improving core utilization on irregular applications through intra-core pipeline parallelism," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020. Available: https://doi.org/10.1109/MICRO50266.2020.00056

[56] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, and B. Steinmacher-Burow, "IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory," *IBM Journal of Research and Development*, vol. 57, no. 1, 2013. Available: https://doi.org/10.1147/JRD.2012.2228092

[57] J. T. Oplinger, D. L. Heine, and M. S. Lam, "In search of speculative thread-level parallelism," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999. Available: https://doi.org/10.1109/PACT.1999.807576

[58] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005, pp. 12 pp.–118. Available: https://doi.org/10.1109/MICRO.2005.13

[59] V. Packirisamy, S. Wang, A. Zhai, W. Hsu, and P. Yew, "Supporting speculative multithreading on simultaneous multithreaded processors," in *High Performance Computing (HiPC)*, Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., 2006. Available: https://doi.org/10.1007/11945918_19

[60] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai, "Exploring speculative parallelism in SPEC2006," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009. Available: https://doi.org/10.1109/ISPASS.2009.4919640

[61] I. Park, B. Falsafi, and T. N. Vijaykumar, "Implicitly-multithreaded processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, A. Gottlieb and K. Li, Eds., 2003. Available: https://doi.org/10.1109/ISCA.2003.1206987

[62] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona, "The Arm Neoverse N1 platform: building blocks for the next-gen cloud-to-edge infrastructure SoC," *IEEE Micro*, vol. 40, no. 2, 2020. Available: https://doi.org/10.1109/MM.2020.2972222

[63] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014. Available: https://doi.org/10.1109/HPCA.2014.6835952

[64] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, K. Pingali, K. A. Yelick, and A. S. Grimshaw, Eds., 2005. Available: https://doi.org/10.1145/1065944.1065964

[65] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8," *Proceedings of the ACM on Programming Languages (POPL)*, vol. 2, no. POPL, pp. 1–29, 2017. Available: https://doi.org/10.1145/3158107

[66] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August, "Decoupled software pipelining with the synchronization array," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004, pp. 177–188. Available: https://doi.org/10.1109/PACT.2004.1342552

[67] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022. Available: https://doi.org/10.1109/MM.2022.3164338

[68] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1997. Available: https://doi.org/10.1109/MICRO.1997.645805

[69] T. B. Schardl, I.-T. A. Lee, J. Carr, D. Curtis, B. Hoppe, C. E. Leiserson *et al.*, "OpenCilk project," https://www.opencilk.org/, Accessed 2023-08-10.

[70] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: embedding fork-join parallelism into LLVM's intermediate representation," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017. Available: https://doi.org/10.1145/3018743.3018758

[71] D. Schor, 2021. Available: https://fuse.wikichip.org/news/6111/intel-details-golden-cove-next-generation-big-core-for-client-and-server-socs/

[72] A. Seznec, "A 256 Kbits L-TAGE branch predictor," https://www.irisa.fr/caps/people/seznec/L-TAGE.pdf, 2007.

[73] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119. Available: https://doi.org/10.1145/1067649.801719

[74] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995. Available: https://doi.org/10.1145/223982.224451

[75] J. G. Steffan, "Hardware support for thread-level speculation," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 2003. Available: https://www.cs.cmu.edu/~./stampede/papers/steffan_phd_thesis.pdf

[76] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems*, vol. 23, no. 3, 2005. Available: https://doi.org/10.1145/1082469.1082471

[77] J. G. Steffan, C. B. Colohan, and T. C. Mowry, *Architectural support for thread-level data speculation*. School of Computer Science, Carnegie Mellon University, 1997.

[78] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: an execution model for fine-grain nested speculative parallelism," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017. Available: https://doi.org/10.1145/3079856.3080218

[79] M. Sung, R. Krashinsky, and K. Asanović, "Multithreading decoupled architectures for complexity-effective general purpose computing," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, p. 56–61, Dec. 2001. Available: https://doi.org/10.1145/563647.563658

[80] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002. Available: https://doi.org/10.1109/MM.2002.997877

[81] I. V. Team, "Extending LoopVectorizer: OpenMP4.5 SIMD and outer loop auto-vectorization," 2016. Available: https://www.llvm.org/devmtg/2016-11/#talk7

[82] T. C. Team, “Clang compiler user manual,” https://clang.llvm.org/docs/UsersManual.html, 2007.

[83] J. Torrellas, *Speculation, thread-level*, D. Padua, Ed. Boston, MA: Springer US, 2011. Available: https://doi.org/10.1007/978-0-387-09766-4_170

[84] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: maximizing on-chip parallelism,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995. Available: https://doi.org/10.1145/223982.224449

[85] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, “Speculative decoupled software pipelining,” in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, 2007, pp. 49–59. Available: https://doi.org/10.1109/PACT.2007.4336199

[86] T. N. Vijaykumar and G. S. Sohi, “Compiling for the multiscalar architecture,” Ph.D. dissertation, School of Computer, Data & Information Sciences, University of Wisconsin - Madison, 1998, aAI9813127.

[87] D. W. Wall, “Limits of instruction-level parallelism,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991. Available: https://doi.org/10.1145/106972.106991

[88] S. Wallace, B. Calder, and D. M. Tullsen, “Threaded multiple path execution,” in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1998. Available: https://doi.org/10.1145/279358.279392

[89] Wikipedia, “Transactional synchronisation extensions (history and bugs),” https://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions#History_and_bugs, Accessed 2024-07-09.

[90] H. Wong, “Measuring reorder buffer capacity,” 2013. Available: https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[91] A. Yasin, “A top-down method for performance analysis and counters architecture,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014. Available: https://doi.org/10.1109/ISPASS.2014.6844459

[92] V. A. Ying, M. C. Jeffrey, and D. Sanchez, “T4: compiling sequential code for effective speculative parallelization in hardware,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020. Available: https://doi.org/10.1109/ISCA45697.2020.00024

[93] A. M. Zaidi, K. Iordanou, M. Luján, and G. Gabrielli, “Loopapalooza: investigating limits of loop-level parallelism with a compiler-driven approach,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021. Available: https://doi.org/10.1109/ISPASS51385.2021.00030

[94] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "BranchNet: a convolutional neural network to predict hard-to-predict branches," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020. Available: https://doi.org/10.1109/MICRO50266.2020.00022