2011 International Conference on Parallel Architectures and Compilation Techniques

Speculative Parallelization in Decoupled Look-ahead

Alok Garg, Raj Parihar, and Michael C. Huang

Dept. of Electrical & Computer Engineering
University of Rochester
Rochester, NY 14627, USA
Email: {garg@urgrad, parihar@ece, michael.huang@}.rochester.edu

Abstract

While a canonical out-of-order engine can effectively exploit implicit parallelism in sequential programs, its effectiveness is often hindered by instruction and data supply imperfections manifested as branch mispredictions and cache misses. Accurate and deep look-ahead guided by a slice of the executed program is a simple yet effective approach to mitigate the performance impact of branch mispredictions and cache misses. Unfortunately, program slice-guided look-ahead is often limited by the speed of the look-ahead code slice, especially for irregular programs. In this paper, we attempt to speed up the look-ahead agent using speculative parallelization, which is especially suited for the task. First, slicing for look-ahead tends to reduce important data dependences that prohibit successful speculative parallelization. Second, the task for look-ahead is not correctness-critical and thus naturally tolerates dependence violations. This enables an implementation to forgo violation detection altogether, simplifying architectural support tremendously. In a straightforward implementation, incorporating speculative parallelization to the look-ahead agent further improves system performance by up to 1.39x with an average of 1.13x.

***Keywords*-Decoupled look-ahead; Speculative parallelization; Helper-threading; Microarchitecture**

I. Introduction

As CMOS technology edging towards the end of roadmap, scaling no longer brings significant device performance improvements. At the same time, increasing transistor budgets are allocated mostly towards increasing throughput and integrating functionality. Without these traditional driving forces, improving single-thread performance for general-purpose applications is without doubt more challenging. Yet, better single-thread performance remains a powerful enabler and is still an important processor design goal.

Out-of-order microarchitecture can effectively exploit inherent parallelism in sequential program, but its capability is often limited by imperfect instruction and data supply: Branch mispredictions inject useless, wrong-path instructions into the execution engine; Cache misses can stall the engine for extended periods of time, reducing effective throughput. Elaborate branch predictors, deep memory hierarchies, and sophisticated prefetching engines are routinely adopted in high-performance processors. Yet, cache misses and branch mispredictions are still important performance hurdles. One general approach to mitigate their impact is to enable *deep* look-ahead so as to overlap instruction and data supply activities with instruction processing and execution.

An important challenge in achieving effective look-ahead is to be deep and accurate at the same time. Simple, state machine-controlled mechanisms such as hardware prefetchers can easily achieve "depth" by prefetching data far into the future access stream. But these mechanisms fall short in terms of accuracy when the access pattern defies simple generalization. On the other hand, a more general form of look-ahead (which we call *decoupled look-ahead*) executes all or part of the program to ensure the control flow and data access streams accurately reflect those of the program. However, such a look-ahead agent needs to be fast enough to provide any substantial performance benefit.

In this paper, we explore speculative parallelization in such a decoupled look-ahead agent. Intuitively, speculative parallelization is aptly suited to the task of boosting the speed of the decoupled look-ahead agent for two reasons. First, the code slice responsible for look-ahead does not contain all the data dependences embedded in the original program, providing more opportunities for speculative parallelization. Second, the execution of the slice is only for look-ahead purposes and thus the environment is inherently more tolerant to dependence violations. We find these intuitions to be largely born out by experiments and speculative parallelization can achieve significant performance benefits at a much lower cost than needed in a general purpose environment.

The rest of this paper is organized as follows: Section II discusses background and related work; Section III details the architectural design; Sections IV and V present the experimental setup and discuss experimental analysis of the design; and Section VI concludes.

II. Background and Related Work

Traditionally uniprocessor microarchitectures have been using look-ahead for a long time to exploit parallelism implicit in sequential codes. However, even for high-end microprocessors, the range of look-ahead is rather limited. Every in-flight instruction consumes some microarchitectural resources and practical designs can only buffer on the orders of 100 instructions. This short "leash" often stalls look-ahead unnecessarily, due to reasons unrelated to look-ahead itself. Perhaps the most apparent case is when a long-latency instruction (*e.g.*, a load that misses all on-chip caches) can not retire, blocking all subsequent instructions from retiring and

This work is supported by an NSF CAREER award CCF-0747324 and also in part by NSF C under the grant 61028004.

releasing their resources. Eventually, the back pressure stalls the look-ahead. This case alone has elicited many different mitigation techniques. For instance, instructions dependent on a long-latency instruction can be relegated to some secondary buffer that are less resource-intensive to allow continued progress in the main pipeline [1], [2]. In another approach, the processor state can be check-pointed and the otherwise stalling cycles can be used to execute in a speculative mode that warms up the caches [1], [3]–[5]. One can also predict the value of the load and continue execution speculatively [6], [7]. Even if the prediction is wrong, the speculative execution achieves some degree of cache warming.

If there are additional cores or hardware thread contexts, the look-ahead effort can be carried out in parallel with the program execution, rather than being triggered when the processor is stalled. Look-ahead can be targeted to specific instructions, such as so-called delinquent loads [8], or can become a continuous process that intends to mitigate all misses and mispredictions. In the former case, the actions are guided by backward slices leading to the target instructions and the backward slices are spawned as individual short threads, often called helper threads or micro threads [8]–[15]. In the latter case, a dedicated thread runs ahead of the main thread. This run-ahead thread can be based on the original program [16]–[20] or based on a reduced version that only serves to prefetch and to help branch predictions [21].

While the two approaches are similar in principle, there are a number of practical advantages of using a single, continuous thread of instructions for look-ahead. First, as shown in Figure 1, the look-ahead thread is an independent thread. Its execution and control is largely decoupled from the main thread. (For notational convenience, we refer to this type of design as decoupled look-ahead.) In contrast, embodying the look-ahead activities into a large number of short helper threads (also known as micro threads) inevitably requires micro-management from the main thread and entails extra implementation complexities. For instance, using extra instructions to spawn helper thread requires modification to program binary and adds unnecessary overhead when the runtime system decides not to perform look-ahead, say when the system is in multithreaded throughput mode.

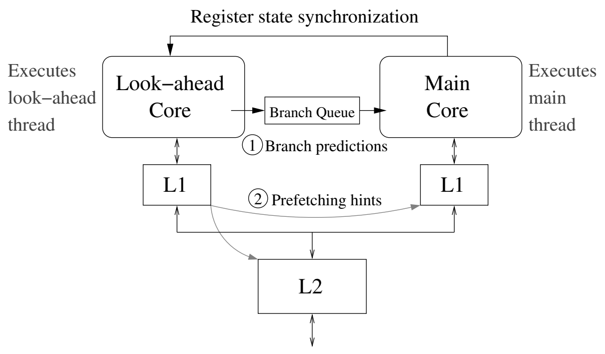


Figure 1. Example of a generic decoupled look-ahead architecture.

Second, prefetching too early can be counter-productive and should be avoided. This becomes an issue when helper threads can also spawn other helper threads to lower the overhead on the main thread [8]. In decoupled look-ahead, since the look-ahead thread pipes its branch outcome through a FIFO to serve as hints to the main thread, it naturally serves as a throttling mechanism, stalling the look-ahead thread before it runs too far ahead. Additionally, addresses can go through a delayed-release FIFO buffer for better just-in-time prefetches.

Finally, as program gets more complex and uses more ready-made code modules, "problematic" instructions will be more spread out, calling for more helper threads. The individual helper threads quickly add up, making the execution overhead comparable to a whole-program based decoupled look-ahead thread. As an illustration, Table I summarizes the statistics about those instructions which are most accountable for last-level cache misses and branch mispredictions. For instance, the top static instructions that generated 90% of the misses and mispredictions in a generic baseline processor accounted for 8.7% (3.6%+5.1%) of total dynamic instruction count. Assuming on average each such instruction instance is being targeted by a very brief 10-instruction long helper thread, the total dynamic instruction count for all the helper threads becomes comparable to the program size. If we target more problematic instance (*e.g.*, 95%), the cost gets even higher.

	Memory references				Branches			
	90%		95%		90%		95%	
	DI	SI	DI	SI	DI	SI	DI	SI
bzip2	1.86	17	3.15	27	3.9	52	4.49	64
crafty	0.73	23	1.04	38	5.33	235	6.14	309
eon	2.28	50	3.34	159	2.02	19	2.31	23
gap	1.35	15	1.44	23	2.02	77	2.64	130
gcc	8.49	153	8.84	320	8.08	1103	8.41	1700
gzip	0.1	6	0.1	6	8.41	40	8.66	52
mcf	13.1	13	14.7	16	9.99	14	10.2	18
parser	1.31	41	1.59	57	6.81	130	7.3	183
pbmk	1.87	35	2.11	52	2.88	92	3.21	127
twolf	2.69	23	3.28	28	5.75	41	6.48	56
vortex	1.96	42	2	67	1.24	114	1.97	167
vpr	7.47	16	11.6	22	4.8	6	4.88	7
Avg	3.60%	36	4.44%	68	5.10%	160	5.56%	236

Table I. Summary of top instructions accountable for 90% and 95% of all last-level cache misses and branch mispredictions. Stats collected on entire run of ref input. DI is the total dynamic instances (measured as a percentage of total program dynamic instruction count). SI is total number of static instructions.

The common purpose of look-ahead is to mitigate misprediction and cache miss penalties by essentially overlapping these penalties (or "bubbles") with normal execution. In contrast, speculative parallelization (or thread-level speculation) intends to overlap normal execution (with embedded bubbles) of different code segments [14], [22]–[27]. If the early execution of a future code segment violates certain dependences, the correctness of the execution is threatened. When a dependence violation happens, usually the speculative execution is squashed. A consequence of the correctness issue is that the architecture needs to carefully track memory accesses in order to detect (potential) dependence violations. In the case of look-ahead execution, by executing the backward slices early, some dependences will also be violated, but the consequence is less severe (*e.g.*, less effective latency hiding). This makes tracking of dependence violations potentially unnecessary (and indeed undesirable as we show later). Because of the non-critical nature of look-ahead thread, exploiting speculative parallelization in the look-ahead context is less demanding. In some applications, notably integer codes represented by SPEC INT benchmarks, the performance of a decoupled look-ahead system is often limited by the speed of the look-ahead thread, making its speculative parallelization potentially profitable.

Note that we proposed to exploit speculative parallelization only in the look-ahead thread. This is fundamentally different from performing look-ahead/run-ahead in a conventional speculative parallelization system [28]. The latter system requires full-blown speculation support that has to guarantee correctness. Comparing the cost-effectiveness of the two different approaches would be an interesting future work.

III. Architectural Design

A. Baseline Decoupled Look-ahead Architecture

Before we discuss the design of software and hardware support to enable speculative parallelization in the look-ahead system, we first describe a baseline decoupled look-ahead system. In this system, a statically generated look-ahead thread executes on a separate core and provides branch prediction and prefetching assistance to the main thread.

1) Basic hardware support: Like many decoupled look-ahead systems [18], [21], [29], our baseline hardware requires modest support on top of a generic multi-core system. Specifically, a FIFO queue is used to pipe the outcome of committed branch instructions to the main thread to be used as the branch predictions. We also pass branch target address for those branches where the target was mispredicted in the look-ahead thread. The FIFO queue contains two bits per branch, one indicates the direction outcome, the other indicates whether a target is associated with this branch. If so, the main thread dequeues an entry from another shallower queue that contains the target address.

The memory hierarchy includes support to confine the state of the look-ahead thread to the L1 cache. Specifically, a dirty line evicted in the look-ahead thread is simply discarded. A cache miss will only return the up-to-date non-speculative version from the main thread. Simple architectural support to allow faster execution of the look-ahead thread is used to reduce stalling due to L2 misses. In some designs, this is the major, if not the only, mechanism to propel the look-ahead thread to run ahead of the main thread [17], [18]. We use the same simple heuristics from [21]: if the look-ahead does not have a sufficient lead over the main thread, we feed 0 to the load that misses in the L2 cache. This is easier to implement than to tag and propagate poison. Finally, prefetching all the way to the L1 is understandably better than prefetching only to L2. We find that a FIFO queue to delay-release the final leg of prefetch (from L2 to L1) is a worthwhile investment as it can significant reduce pollution.

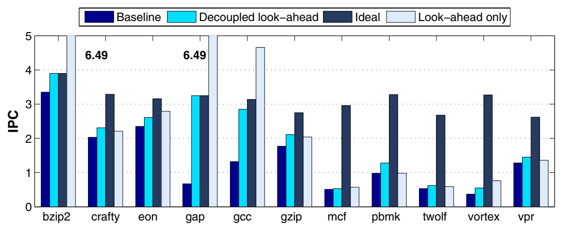
2) Look-ahead thread generation: For the software, we use an approach similar to the one proposed by Garg and Huang [21] where a binary analyzer creates a skeleton out of the original program binary just for look-ahead. Specifically, biased branches (with over 99.9% bias towards one direction) are converted into unconditional branches. All branches and their backward slices are included on the skeleton. Memory dependences are profiled to exclude long-distance dependences: if a load depends on a store that consistently (more than 99.9%) has long def-to-use distance (> 5000 instructions) in the profiling run, backward slicing will terminate at the load: these stores are too far away from the consumer load that even if they are included in the look-ahead thread, the result would likely be evicted from the L1 cache before the load executes. Finally, memory instructions that often miss the last-level cache that are not already included in the skeleton are converted to prefetch instructions and added to the skeleton. All other instructions on the original program binary become NOPs.

3) Code address space and runtime recoveries: The resulting skeleton code body has the same basic block structure and branch offsets. Therefore, the two code segments (original code and the skeleton) can be laid out in two separate code address spaces or in the same code address space (which will leave a fixed offset between any instruction in the skeleton and its original copy in the original code). In either case, a simple support in the instruction TLB can allow two threads to have the exact same virtual addresses and yet fetch instructions from different places in the main memory. As such, when the main thread is being context switched-in, the same initialization can be applied to its look-ahead thread. Similarly, when the branch outcomes from the look-ahead threads deviate from that of the main thread, it can be "rebooted" with a checkpoint register state from the main thread. We call such an event a *recovery*.

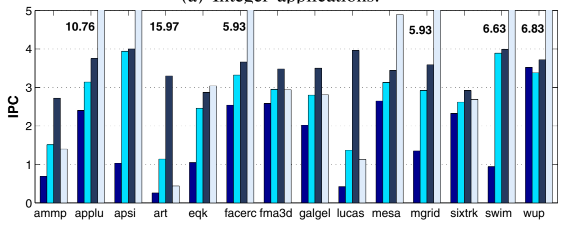
4) Motivation for look-ahead thread parallelization: With two threads running at the same time, the performance is dictated by whichever runs slower. If the look-ahead thread is doing a good enough job, the speed of the duo will be determined by how much execution parallelism can be extracted by the core. The performance potential can be approximated by idealizing cache hierarchy and branch prediction. Note that idealization tends to produce a loose upper-bound. On the other hand, when the look-ahead thread is the bottleneck, the main thread can only be accelerated up to the speed of the look-ahead thread running alone, which is another performance upper-bound. While the details of the experimental setup will be discussed later, Figure 2 shows the performance of a baseline decoupled look-ahead system described above measured against the two upper-bounds just discussed. As a reference point, the performance of running the original binary on a single core in the decoupled look-ahead system is also shown.

The applications can be roughly grouped into three categories. First, for 9 applications (*bzip*, *gap*, *gcc*, *apsi*, *facerec*, *mesa*, *sixtrack*, *swim*, and *wupwise*), the baseline look-ahead is quite effective, allowing the main thread to sustain IPC close to that in the ideal environment (within 10%). Further improving the speed probably requires at least making the core wider.

For a second group of 2 applications (*applu*, and *mgrid*), the absolute performance is quite high (with IPCs around 3) for the baseline decoupled look-ahead system but there is still some performance headroom against the ideal environment. However, the speed of the look-ahead thread does not appear to be the problem as it is running fast enough on its own. There are a range of imperfections in the entire system that are the source of the performance gap. For example, the quality of information may not be high enough to have complete coverage of the misses. The look-ahead thread's L1

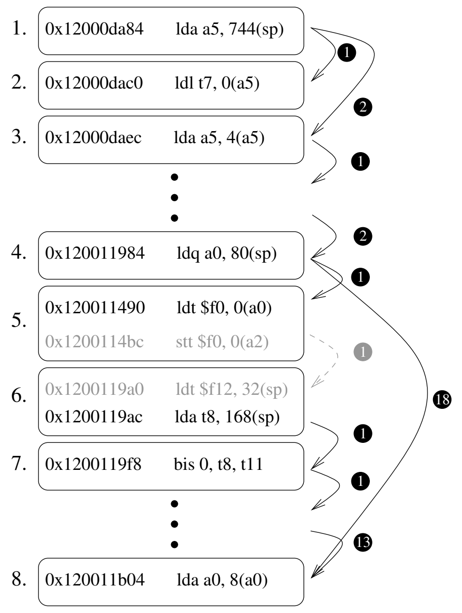


(a) Integer applications.



(b) Floating-point applications.

Figure 2. Comparison of performance of a baseline core, a decoupled look-ahead execution, and the two upper-bounds: a baseline with idealized branch prediction and memory hierarchy, and look-ahead thread running alone. Since the look-ahead thread has many NOPs that are removed at the pre-decode stage, their effective IPC can be higher than the pipeline width. Also note that because the baseline decoupled look-ahead execution can skip through some L2 misses at runtime, the speed of running look-ahead thread alone (but without skipping any L2 misses) is not a strict upper bound, but an approximation.



```
1. 0x12000da84   lda a5, 744(sp)
2. 0x12000dac0   ldl t7, 0(a5)
3. 0x12000daec   lda a5, 4(a5)
4. 0x120011984   ldq a0, 80(sp)
5. 0x120011490   ldt $f0, 0(a0)
   0x1200114bc   stt $f0, 0(a2)
6. 0x1200119a0   ldt $f12, 32(sp)
   0x1200119ac   lda t8, 168(sp)
7. 0x1200119f8   bis 0, t8, t11
8. 0x120011b04   lda a0, 8(a0)
```

Figure 3. Example of the change of dependence characteristics due to skeleton construction. In this example from application *equake*, each box represents a basic block. For clarity, only a subset of instructions are shown. Instructions not on the skeleton are shown in gray color. The arcs show the dependences and their distance measured in numbers of basic blocks.

cache may hold stale data causing the thread to veer off the actual control flow causing recoveries.

For the remaining 14 applications, the application's speed is mainly limited by the speed of the look-ahead thread, indicating potential performance gain when the look-ahead thread is accelerated.

B. New Opportunities for Speculative Parallelization

There are two unique opportunities in the look-ahead environment to apply speculative parallelization. First, the construction of the skeleton removes some instructions from the original binary and thereby removes some dependences. Our manual inspection of a few benchmarks finds a repeating pattern of complex, loop-carried dependences naturally removed as the skeleton is being constructed, giving rise to more loop-level parallelism. Another pattern is the increase in dependence distance as a result of removing instructions in short-distance dependence chains. As shown in the example in Figure 3, in the original code, almost every basic block depends on its immediate predecessor basic block. When constructing the skeleton, the removal of some instructions from the basic blocks 5 and 6 breaks the chain of dependences, leaving only a long-distance dependence which provides exploitable parallelism.

A second opportunity is the lowered requirement for speculative parallelization. When two sequential code sections A and B are executed in parallel speculatively, register and memory dependences that flow from A to B are preserved by a combination of explicit synchronization and squashing of instructions detected to have violated the dependence. Because register dependences are explicit in the instruction stream, explicit synchronization can be used to enforce them. Nevertheless, the extra synchronization adds to the overhead of execution and demands special hardware support. For memory dependences, without complete static knowledge of the exact addresses, compiler can not identify all possible dependences and run-time tracking becomes necessary.

In contrast, when we try to exploit speculative parallelization in the inherently non-critical look-ahead thread, correctness is no longer a must, but rather a quality of service issue: unenforced dependences only *presumably* reduce the effectiveness of look-ahead and affect performance, and may not be worth fixing. Indeed, fixing a dependence violation using rollback slows down the look-ahead and thus may cause more detriment than the dependence violation. Thus, it becomes a matter of choice how much hardware and runtime support we need to detect violations and repair them. In the extreme, we can forgo all conventional hardware support for speculative parallelization and only rely on probabilistic analysis of the dependence to minimize violations.

Finally, in hindsight, we realized that speculative parallelization of the look-ahead thread also has a (small) side effect of reducing the cost of recoveries. A recovery happens when the branch outcome from the look-ahead thread deviates from that of the main thread. For simplicity, we reboot the look-ahead thread rather than try to repair the state. Because the look-ahead threads can be running far ahead of the main thread, such a recovery can wipe out the lead the look-ahead thread accumulated over a period of time. Spawning a secondary thread provides a natural mechanism to preserve part of the look-ahead that has already been done as we can reboot one thread while keep the other running. We will discuss this and present quantitative results in more detail later.

C. Software Support

1) Dependence analysis: To detect coarse-grain parallelism suitable for thread-level exploitation, we use a profile guided analysis tool. The look-ahead thread binary is first profiled to identify dependences and their distance. To simplify the subsequent analysis, we collapse the basic block into a single node, and represent the entire execution trace as a linear sequence of these nodes. Dependences are therefore between basic blocks and the distance can be measured by the distance of nodes in this linear sequence as shown in Figure 4-(a).

Given these nodes and arcs representing dependences between them, we can make a cut before each node and find the minimum dependence distance among all the arcs that pass through the cut. This minimum dependence distance, or D_{min}, represents an approximation of parallelization opportunity as can be explained by the simple example in Figure 4. Suppose, for the time being, that the execution of a basic block takes one unit of time and there is no overlapping of basic block execution. Node d in Figure 4, which has a D_{min} of 3, can therefore be scheduled to execute 2 units of time ($D_{min} - 1$) earlier than its current position in the trace – in parallel with node b. All subsequent nodes can be scheduled 2 units of time earlier as well, without reverting the direction of any arcs.

Of course, the distance in basic blocks is only a very crude estimate of the actual time lapse between the execution of the producer and consumer instructions.[1] In reality, the size and execution speed of different basic blocks is different and their executions overlap. Furthermore, depending on the architectural detail, executing the producer instructions earlier than the consumer does not necessarily guarantee the result will be forwarded properly. Therefore, for nodes with small D_{min} there is little chance to exploit parallelism. We set a threshold on D_{min} (in Figure 4) to find all candidate locations for a spawned thread to start its execution. D_{min} threshold in our study is 15 basic blocks which is approx. 120 instructions.

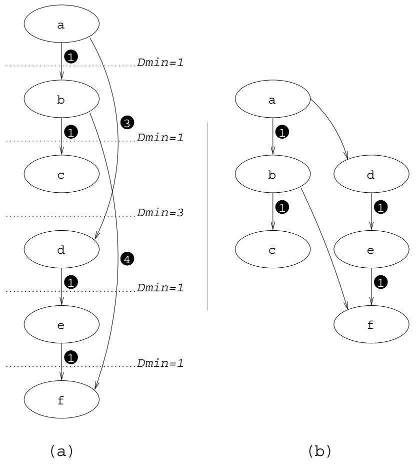


Figure 4. Example of basic-block level parallelization.

[1]A somewhat more appropriate notion of dependence distance that we use is the actual number of instructions in between the source and destination basic blocks.

2) Selecting spawn and target points: With the candidates selected from profile-based analysis, we need to find static code locations to spawn off parallel threads (*e.g.*, node a in Figure 4-b), and locations where the new threads can start their execution (*e.g.*, node d in Figure 4-b). We call the former *spawn points*, the latter *target points*. We first select target points. We choose those static basic blocks whose dynamic instances consistently show a D_{min} larger than the threshold value.

Next, we search for the corresponding spawn points. The choices are numerous. In the example shown in Figure 4, if we ignore the threshold of D_{min}, the spawn point of node d can be either a or b. Given the collection of many different instances of a static target point, the possibilities are even more numerous. The selection needs to balance cost and benefit. In general, the more often a spawn is successful and the longer the distance between the spawn and target points the more potential benefit there is. On the other hand, every spawn point will have some unsuccessful spawns, incurring costs. We use a cost benefit ratio ($\Sigma distances/\#false\ spawns$) to sort and select the spawn points. Note that a target point can have more than one spawn points.

3) Loop-level parallelism: Without special processing, a typical loop using index variable can project a false loop-carried dependence on the index variable and masks potential parallelism from our profiling mechanism. After proper adjustments, parallel loops will present a special case for our selection mechanism. The appropriate spawn and target points would be the same static node. The number of iterations to jump ahead is selected to make the number of instructions close to a target number (1000 in this paper).

4) Available parallelism: A successful speculative parallelization system maximizes the opportunities to execute code in parallel (even when there are apparent dependences) and yet does not create too many squashes at runtime. In that regard, our methodology has a long way to go. Our goal in this paper is to show that there are significant opportunities for speculative parallelization in the special environment of look-ahead, even without a sophisticated analyzer. Figure 5 shows an approximate measure of available parallelism recognized by our analysis mechanism. The measure is simply that of the "height" of a basic block schedule as shown in Figure 4. For instance, the schedule in Figure 4-a has a height of 6 and the schedule in Figure 4-b has a height of 4, giving a parallelism of 1.5 (6/4). (Note that in reality, node d's D_{min} is too small to be hoisted up for parallel execution.) For simplicity, at most two nodes can be in the same time slot, giving a maximum parallelism of 2. For comparison, we also show the result of using the same mechanism to analyze the full program trace.

The result shows that there is a significant amount of parallelism, even in integer code. Also, in general, there is more parallelism in the look-ahead thread than in the main program thread.

Figure 5-(b) shows the amount of parallelism when we impose further constraints, namely finding stable spawn-target point pairs to minimize spurious spawns. The result suggests that some parallelism opportunities are harder to extract than others because there are not always spawn points that consistently lead to the execution of the target point. We

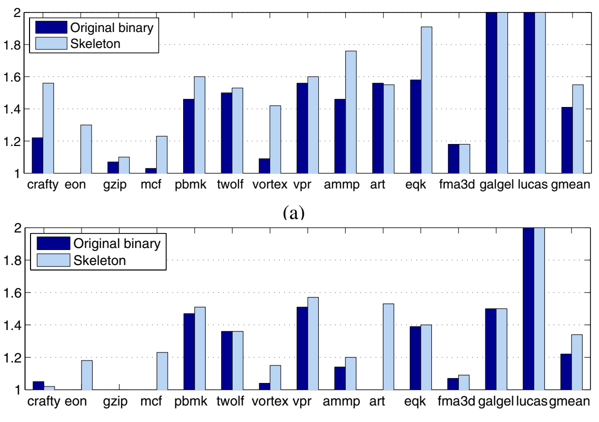


Figure 5. An approximate measure of available parallelism in the trace of the look-ahead thread and the main program thread (a). A more constrained measure of parallelism that is likely to be exploited (b).

leave it as future work to explore cost-effective solutions to this problem.

D. Hardware Support

Typical speculative parallelization requires a whole host of architectural support such as data versioning and cross-task dependence violation detection [30]. Since look-ahead does not require correctness guarantee, we are interested in exploring a design that minimizes intrusion.

To enable the speculative parallelization discussed so far, there are three essential elements: we need to (1) spawn a thread, (2) communicate values to the new thread, and (3) properly merge the threads.

1) Spawning a new thread: The support we need is not very different from existing implementation to spawn a new thread in a multi-threaded processor. For example, the main thread will set up the context of the newly spawned thread (Figure 6). A key difference is that the spawned thread is executing a future code segment in the same logical thread. If everything is successful, the primary look-ahead thread is expected to reach where the spawned thread has already started and "merge" with the spawned thread (Figure 6). Therefore, the starting PC of the newly spawned thread needs to be recorded. When the primary thread reaches that same PC – more specifically, when the instruction under that PC is about to retire – the primary thread simply terminates, without retiring that instruction, since it has been executed by the spawned thread.

2) Support for register access: When a new thread is spawned, it inherits the architectural state including memory content and register state. While this can be implemented in a variety of environments, it is most straightforward to support in a multithreaded processor. We focus on this environment.

When the spawning instruction is dispatched, the register renaming map is duplicated for the spawned thread. With this action, the spawned thread is able to access register results defined by instructions in the primary thread prior to the spawn point. Note that if an instruction (with a destination register of say, $R2$) prior to the spawn point has yet to execute, the issue logic naturally guarantees that any subsequent instruction depending on $R2$ will receive

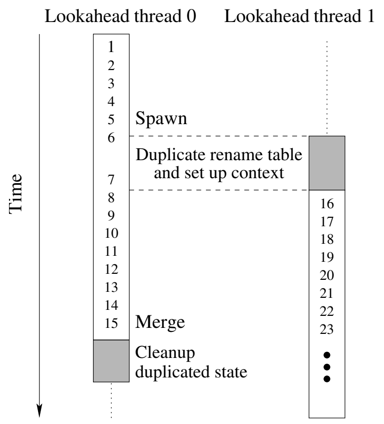


Figure 6. Illustration of runtime thread spawn and marge.

the proper value regardless of which thread the instruction belongs to (*e.g.*, register r6 in Figure 7).

When a rename table entry is duplicated, a physical register is mapped in two entries and both entries can result in the release of the register in the future. A single bit per rename table entry is therefore added to track "ownership" ('O' bit in Figure 7). When the spawned thread copies the rename table, the ownership bits are set to 0. A subsequent instruction overwriting the entry will not release the old register, but will set the ownership bit to 1 (*e.g.*, in Figure 7, 'O' bit of register r6 in spawned thread's map table is set to 1 after renaming).

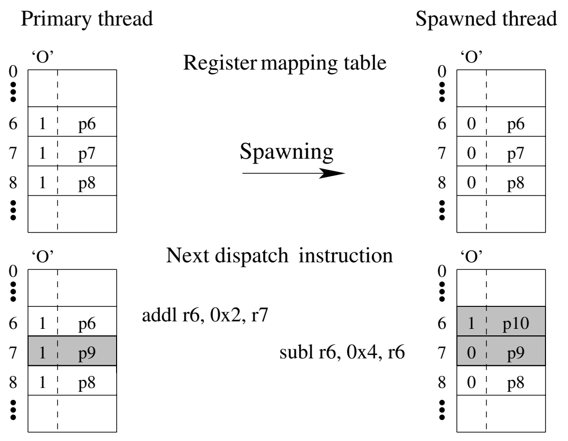


Figure 7. Register renaming support for the spawned thread. Entry 'O' in map tables refers to "ownership" bit.

Since the bit indicates whether the spawned thread has defined its own version of a particular architectural register, a 0 means the corresponding register should carry the last update to that architectural register made by the primary thread. This is approximated by updating the rename mapping of the architectural register as the primary thread updates its corresponding entry. This is illustrated in Figure 7, where update of $r7$ in the primary thread changes both threads' $r7$ mapping to $p9$. In other words, after the thread is spawned, it may still receive register inputs from the primary thread. A slightly simpler alternative is not to support such inputs. We find that to be also working fine in the vast majority of cases with negligible performance impact, but it does result in a significant (16%) performance degradation in one application.

Finally, when the primary thread terminates (at merge point), any physical register that is not mapped in the secondary thread's rename table can be recycled. This can be easily found out from the ownership bit vector: any bit of 1 indicates the threads both have their thread-local mapping and thus the register is private to the primary thread and can be recycled.

3) Support for memory access: Getting memory content from the primary thread is also simplified in the multi-threaded processor environment since the threads share the same L1 cache. An extreme option is to not differentiate between the primary look-ahead thread and the spawned thread in cache accesses. This option incurs the danger that write operations to the same memory location from different threads will not be differentiated and subsequent reads will get wrong values. However, even this most basic support is a possible option, though the performance benefit of parallel look-ahead is diminished as we will show later.

A more complete versioning support involves tagging each cache line with thread ID and returning the data with the most recent version for any request. For conventional speculative parallelization, this versioning support is usually done at a fine access granularity to reduce false dependence violation detections [31]. In our case, we use a simplified, coarse-grain versioning support without violation detection, which is a simple extension of the cache design in a basic multithreaded processor. For notational convenience we call this *partial* versioning.

The main difference from a full-blown versioning support is two-fold. First, version is only attached to the cache line as in the baseline cache in a multi-threaded processor. No per-word tracking is done. Similar to versioning cache, a read from thread i returns the most recent version no later than i. A write from thread i creates a version i from the most recent version if version i does not already exist. The old version is tagged (by setting a bit) as being replaced by a new version. This bit is later used to gang-invalidate replaced lines. Second, no violation detection is done. When a write happens, it does not search for premature reads from a future thread. The cache therefore does not track whether any words in a line have been read.

E. Runtime Spawning Management

Based on the result of our analysis and to minimize unnecessary complexities, we opt to limit the number of threads spawned at any time to only one. This simplifies hardware control such as when to terminate a running thread and partial versioning support. It also simplifies the requirement on dependence analysis.

At run-time, two thread contexts are reserved (in a multithreaded core) for look-ahead. There is always a primary thread. A spawn instruction is handled at dispatch time and will freeze the pipeline front end until the rename table is duplicated and a new context is set up. If another thread is already occupying the context, the spawn instruction is discarded. Since the spawn happens at dispatch, it is a speculative action and is subject to a branch misprediction squash. Therefore we do not start the execution immediately, but wait for a short period of time. This waiting also makes it less likely that a cross-thread read-after-write dependence is violated. When the spawn instruction is indeed squashed due to branch misprediction, we terminate the spawned thread.

When the primary thread reaches the point where the spawned thread started execution, the two successfully merge. The primary thread is terminated and the context is available for another spawn. The spawned thread essentially carries on as the primary look-ahead thread. At this point, we gang invalidate replaced cache lines from the old primary thread and consolidate the remaining lines into the new thread ID.

When the primary thread and the spawned thread deviate from each other, they may not merge for a long time. When this happens, keeping the spawned thread will prevent new threads from being spawned and limit performance. So run-away spawns are terminated after a fixed number of instructions suggested by the software.

F. Communicating Branch Predictions to Primary Thread

Branch predictions produced by look-ahead thread(s) are deposited in an ordered queue called branch queue. There are many options to enforce semantic sequential ordering among branch predictions despite that they might be produced in different order. One of the simpler options we explored in this paper is to segment the branch queue in a few banks of equal size as shown in Figure 8.

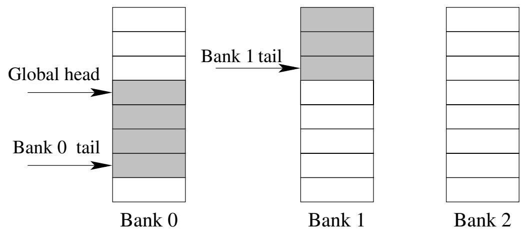


Figure 8. Example of a banked branch queue. Bank 0 and bank 1 are written by primary (older) look-ahead and spawned (younger) look-ahead threads respectively. Primary thread uses global head pointer, currently pointing to an entry in bank 0, to read branch predictions.

The bank management is straightforward. When a thread is spawned a new bank is assigned in sequential order. A bank is also de-allocated sequentially as soon as the primary thread consumes all branch predictions form that bank. If a thread exhausts the entries in its current bank, the next sequential bank is used. It is possible, but very rare, that the next bank has already been allocated to a spawned thread. In such a case, to maintain the simplicity of sequential allocation, we kill the younger thread and reclaim bank for re-assignment.

IV. Experimental Setup

We perform our experiments using an extensively modified version of SimpleScalar [32]. Support was added for Simultaneous Multi-Threading (SMT), decoupled look-ahead, and speculative parallelization. Because of the approximate nature of the architecture design for look-ahead, the result of the semantic execution is dependent on the microarchitectural state. For example, a load from the look-ahead thread does not always return the latest value stored by that thread because that cache line may have been evicted. Therefore, our modified simulator uses true execution-driven simulation where values in the caches and other structures are faithfully modeled. The values are carried along with instructions in the pipeline and their semantic execution are emulated on the fly to correctly model the real execution flow of the look-ahead thread.

1) Microarchitecture and configuration: The simulator is also enhanced to faithfully model issue queues, register renaming, ROB, and LSQ. Features like load-hit speculation (and scheduling replay), load-store replays, keeping a store miss in the SQ while retiring it from ROB are all faithfully modeled [33]. We also changed the handling of prefetch instructions (load to ZERO register – R31). By default, the original simulator not only unnecessarily allocates an entry in the LQ, but fails to retire the instruction immediately upon execution as indicated in the alpha processor manual [33]. In our simulator, a prefetch neither stalls nor takes resource in the LQ. Our baseline core is a generic out-of-order microarchitecture loosely modeled after POWER5 [34]. Details of the configurations are shown in Table II.

Baseline core	
Fetch/Decode/Commit	8 / 4 / 6
ROB	128
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div
Issue Q / Reg. (int,fp)	(32, 32) / (120, 120)
LSQ(LQ,SQ)	64 (32,32) 2 search ports
Branch predictor	Bimodal + Gshare
- Gshare	8K entries, 13 bit history
- Bimodal/Meta/BTB	4K/8K/4K (4-way) entries
Br. mispred. penalty	at least 7 cycles
L1 data cache	32KB, 4-way, 64B line, 2 cycles, 2 ports
L1 I cache (not shared)	64KB, 1-way, 128B, 2 cyc
L2 cache (uni. shared)	1MB, 8-way, 128B, 15 cyc
Memory access latency	400 cycles

Table II. Core configuration.

An advanced hardware-based global stream prefetcher based on [35], [36] is also implemented between the L2 cache and the main memory: On an L2 miss, the stream prefetcher detects an arbitrarily sized stride by looking at the history of past 16 L2 misses. If the stride is detected twice in the history buffer, an entry is allocated on the stream table and prefetch is generated for the next 16 addresses. Stream table can simultaneously track 8 different streams. For a particular stream, it issues a next prefetch only when it detects the use of previously prefetched cache line by the processor.

2) Applications and inputs: We use SPEC CPU2000 benchmarks compiled for Alpha. We use the *train* input for profiling, and run the applications to completion. For evaluation, we use *ref* input. We simulate 100 million instructions after skipping over the initialization portion as indicated in [37].

V. Experimental Analysis

We first look at the end result of using our speculative parallelization mechanism and then show some additional experiments that shed light on how the system works and point to future work to improve the design's efficacy.

A. *Performance analysis*

Recall that a baseline look-ahead system can already help many applications execute at a high throughput that is close to saturating the baseline out-of-order engine. For these applications, the bottleneck is not the look-ahead mechanism. Designing efficient wide-issue execution engine or look-ahead to assist speculative parallelization of the main thread are directions to further improve their performance. For the remaining applications, Figure 9 shows the relative performance of a baseline, single-threaded look-ahead system and our speculative, dual-threaded look-ahead system. All results are normalized to that of the single-core baseline system.

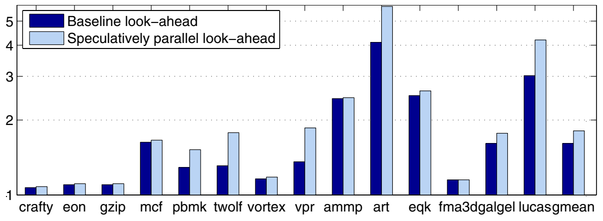


Figure 9. Speedup of baseline look-ahead and speculatively parallel look-ahead over single-core baseline architecture. Because the uneven speedups, the vertical axis is log-scale.

Our speculative parallelization strategy provides up to 1.39x speedup over baseline look-ahead. On average, measured against the baseline look-ahead system, the contribution of speculative parallelization is a speedup of 1.13. As a result, the speedup of look-ahead over a single core improves from 1.61 for single look-ahead thread to 1.81 for two look-ahead threads. If we only look at the integer benchmarks in this set of applications, traditionally considered harder to parallelize, the speedup over baseline look-ahead system is 1.11.

It is worth noting that the quantitative results here represent what our current system allows us to achieve. It does not represent what could be achieved. With more refinement and trial-and-error, we believe more opportunities can be explored. Even with these current results, it is clear that speeding up sequential code sections via decoupled look-ahead is a viable approach for many applications.

Finally, for those applications where the main thread has (nearly) saturated the pipeline, this mechanism does not slow down the program execution. The detailed IPC results for all applications are shown in Table III.

B. *Comparison with conventional speculative parallelization*

As discussed earlier, the look-ahead environment offers a unique opportunity to apply speculative parallelization technology partly because the code to drive look-ahead activities removes certain instructions and therefore provide more potential for parallelism. However, an improvement in the speed of the look-ahead only indirectly translates into end performance. Here, we perform some experiments to inspect the impact.

We use the same methodology on the original program binary and support the speculative threads to execute on a multi-core system. Again, our methodology needs further improvement to fully exploit available parallelism. Thus the absolute performance results are almost certainly underestimating the real potential. However, the relative comparison can still serve to contrast the difference in the two setups.

Figure 10 shows the results. For a more relevant comparison, conventional speculative parallelization is executed on two cores to prevent execution resource from becoming the bottleneck. It is worth nothing, the base of normalization is not the same. For the conventional system, the speedup is over a single core running the original program (A). For our

	bzip2	crafty	eon	gap	gcc	gzip	mcf	pbmk	twolf	vortex	vpr	ammp	applu	apsi	art	equake	fac	fma3d	galgel	lucas	mesa	mgrid	six	swim	wup
1	1.32	2.30	2.63	1.92	2.20	2.14	0.51	0.89	0.57	1.93	1.31	0.79	1.81	1.75	0.27	1.09	3.03	2.72	2.35	0.58	2.99	3.03	2.84	1.26	3.77
2	1.75	2.47	2.90	3.35	4.60	2.34	0.83	1.14	0.74	2.24	1.77	1.92	2.76	2.34	1.13	2.73	3.65	3.12	3.79	1.75	3.36	3.83	3.12	3.78	4.11
3	1.75	2.48	2.91	3.36	4.81	2.36	0.84	1.35	1.01	2.27	2.43	1.93	2.75	2.49	1.57	2.85	3.68	3.12	4.17	2.44	3.37	3.83	3.12	3.78	4.11

Table III. IPC of baseline (1), baseline look-ahead (2), and speculatively parallel look-ahead (3). Note that all prefetch instructions in the main thread are superfluous in our system and are assumed to be eliminated in predecode stage and do not consume resources.

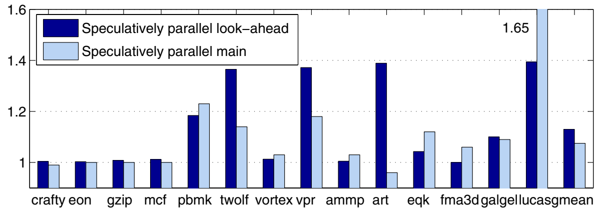


Figure 10. Comparison of the effect of our speculative parallelization mechanism on the look-ahead thread and on the main thread.

system, the speedup is over a baseline look-ahead system running a sequential look-ahead thread (B). B has much higher absolute performance than A.

As we showed earlier using a simple model of potential parallelism (Figure 5), there is more parallelism in the look-ahead binary (the skeleton) than in the full program binary. In Figure 10, we see that in many cases, this translates into more performance gain in the end for the look-ahead system. However, there are phases of execution where the look-ahead speed is not the bottleneck. A faster look-ahead thread only leads to filling the queues faster. Once these queues that pass information to the main thread fill up, look-ahead stalls. Therefore, in some cases, the advantage in more potential parallelism does not translate into more performance gain.

C. System Diagnosis

1) Recoveries: When the look-ahead thread's control flow deviates from that of the main thread, the difference in branch outcome eventually causes a recovery. The ability of the look-ahead to run far ahead of the main thread is crucial to performing useful help. This ability is a direct result of approximations such as ignoring uncommon cases, which come at the price of recoveries. Speculative parallelization in the look-ahead environment also introduces its own sets of approximations. Otherwise, the opportunities will be insufficient and the implementation barrier will be too high. Certainly, too much corner-cutting can be counter-productive. Table IV summarizes the maximum, average, and minimum recovery rate for integer and floating-point applications.

	INT			FP		
	Max	Avg.	Min	Max	Avg.	Min
Baseline look-ahead	3.21	1.24	0.05	2.87	0.34	0.00
Spec. parallel look-ahead	6.55	1.21	0.05	2.83	0.34	0.00

Table IV. Recovery rate for baseline and speculatively parallel look-ahead systems. The rates are measure by number of recoveries per 10,000 committed instructions in the main thread.

For most applications, the recovery rate stays essentially the same. For some applications the rate actually reduces (*e.g.*, *perlbmk*, from 3.21 to 0.43). Recall that skipping an L2 miss (by returning a 0 to the load instruction) is an effective approach to help the look-ahead thread stay ahead of the main thread. Frequent applications of this technique inevitably increase recoveries. In our system, this technique is only applied when the trailing main thread gets too close. With speculative parallelization, the need to use this technique decreases, as does the number of resulting recoveries.

2) Partial recoveries: As discussed earlier, when a recovery happens, we reboot only the primary look-ahead thread. If there is another look-ahead thread spawned, we do not terminate the spawned thread, even though the recovery indicates that some state in the look-ahead threads is corrupted. We have this option because the look-ahead activities are not correctness critical. This essentially allows a partial recovery (without any extra hardware support) and maintains some lead of look-ahead. Nevertheless, it is possible that the spawned thread is corrupt and this policy only delays the inevitable. Table V shows that this is not the case. The first row of numbers indicate how often a recovery happens when both look-ahead threads are running. In some of these cases, the rebooting of the primary thread may send it down a different control flow path so it can no longer merge with the spawned thread. In the rest of the cases (the majority in our experiments as indicated by the second row of the table), the merge happens successfully despite the reboot of the primary thread.

Still, the corruption exists and can cause latent errors that triggers a recovery soon down the road. The next two rows show how often the spawned thread is still live (has not encountered its recovery) 200 and 1000 instructions after the merge point. In most applications, a large majority of instances of the spawned thread are alive and well past 1000 instructions, indicating that indeed they deserved to be kept alive at the recovery point. Also, the difference between the number of instances alive at 200 and 1000 instructions point is small, indicating that those that do not survive long actually terminate rather early. All in all, it is clear that keeping these spawned threads live has low risks.

	gap	mcf	pbm	twf	vor	vpr	amp	eqk	fac	fma	msa	wup
Recv-Both	44	91	20857	37774	2280	5804	1938	7396	415	2030	231	6008
Recv-Merge	38	91	6085	30234	1460	4997	1185	6380	409	1992	226	6008
Live-200	36	91	4099	22633	1446	4855	933	2617	71	1985	224	6008
Live-1000	32	90	1480	10821	1433	4457	760	2025	69	1982	220	6008

Table V. Partial recoveries in speculative look-ahead. Recv-Both are the number of total recoveries when both look-ahead threads are running. Recv-Merge are the instances when after the reboot of the primary look-ahead thread, it successfully merges with the spawned thread which is not rebooted. Out of these instances, Live-200 and Live-1000 are those where the spawned thread is still live (no recovery of their own) 200 and 1000 instructions respectively post merge.

Of course, these numbers can only suggest that not killing an already-spawned thread during recovery *may* be justifiable. In reality, each case is different, and a more discerning policy may be a better choice than a fixed policy. In our simulations, we found that consistently keeping the spawned look-ahead thread alive performs about 1% better *on average* than always killing it at recovery. However, two applications demonstrate the opposite behavior where always killing the spawned thread is the better choice.

3) Spawns: Table VI shows statistics about the number of spawns in different categories. The top half shows the cases where a spawn happens on the right path. They are predominantly successful. Only a handful of them become runaway spawns (not merging with primary thread after a long time).

	mcf	pbm	twf	vor	vpr	ammp	art	eqk	fma	gal	lucas
Spawns invoked under correct path											
Successful	2297	26873	21067	1273	42082	6328	29598	16676	9687	20997	24022
Runaway	257	245	1738	37	409	3542	363	0	3965	0	1
Spawns invoked under incorrect path											
No disp.	11	707	2837	96	1633	26	29	245	363	1	0
Some disp.	28	69	1803	6	273	45	116	10	1	0	0
WP	11	184	2997	152	111	339	6	62	4	17	0

Table VI. Breakdown of all the spawns. The top half shows spawns on the right path, which either successfully merged or were killed because they become runaway spawns. The bottom half shows all spawns on the wrong path – either predicted wrong path (fixed by branch execution) or committed wrong path (fixed by recovery). In the former case, the spawn may have dispatched no instruction or some instructions by the time the branch is resolved. For crafty, eon, and gzip we do not see any spawns in our experiments.

The bottom half shows spawns on the wrong path due to branch mispredictions of the look-ahead thread or because the primary look-ahead thread deviate from the right control flow. All these spawns are wasteful, spurious spawns. We can see that their numbers are much smaller than successful spawns. Furthermore, recall that the spawned thread does not execute immediately after the spawn, but wait for a small period of time (to minimize unnecessary execution due to branch misprediction-triggered spawns and also to reduce violation of dependence). As a result of this small delay, in many cases (row labeled "No disp."), the waste is small as spuriously spawned threads have not dispatched any instruction before the branch is resolved and the spawn squashed. Almost all of these spurious spawns are short lived even for those cases where some instructions on the spawned thread have been dispatched. In summary, speculative parallelization does not significantly increase the energy cost as the waste is small. Our speculatively parallel look-ahead system executes on average 1.5% more instructions than sequential look-ahead due to very few failed spawns.

4) Effect of speculation support: Because look-ahead thread is not critical for correctness, supporting speculative parallelization can be a lot less demanding than otherwise – in theory. In practice, there is no appeal for complexity reduction if it brings disproportionate performance loss. Section III-D described a design that does not require full-blown versioning and has no dependence violation tracking.The cache support required is a much more modest extension of cache for multithreaded core. In Figure 11, we compare this design to one that is even more relaxed: the data cache has no versioning support and in fact is completely unaware of the distinction between the primary look-ahead thread and the spawned one. As the figure shows, some applications (*e.g.*, *twolf*) suffer a noticeable performance loss. However, for most applications the degradation is negligible. The impact on average is very small. Since L1 caches are critical for performance and is often the subject of intense circuit timing optimization, any significant complication can create practical issues in a high-end product. Speculative parallelization in look-ahead, however, gives the designers the capability to choose incremental complexity with different performance benefits, rather than an all-or-nothing option as in conventional speculative parallelization.

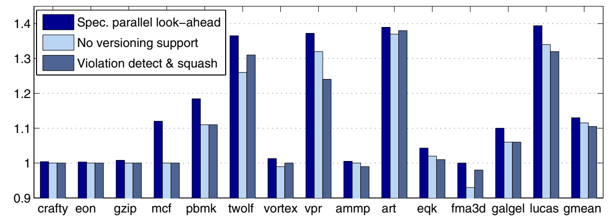


Figure 11. Speedup comparison of regular support and two other alternatives, one removing the partial versioning support altogether, the other adding dependence violation detection to squash the spawned thread.

Another example of the design flexibility is about whether to detect dependence violations. Dependence violation detection also requires intrusive modifications. The flexibility of not having to support it is thus valuable. Figure 11 also compares our design to another one where accesses are carefully tracked and when a dependence violation happens, the spawned thread is squashed. This policy provides no benefit in any application window we simulated and degraded performance significantly in several cases (*e.g.*, *perlbmk* and *vpr*). Intuitively, the look-ahead process is somewhat error tolerant. Being optimistic by ignoring the occasional errors is not only easier to support but better in effects.

VI. Conclusions

Improving single-thread performance for general-purpose programs continues to be an important goal in the design of microprocessors. Executing dedicated code to run ahead of program execution to help mitigate performance bottlenecks from branch mispredictions and cache misses is a promising approach. A particularly straightforward implementation, which we call decoupled look-ahead runs an independent, program-length thread to achieve performance enhancement. In some cases, the approach allows the program to nearly saturate a high-performance out-of-order pipeline. In many other cases, the speed of the look-ahead thread becomes the bottleneck.

In this paper, we have proposed a mechanism to apply speculative parallelization to the look-ahead thread. This approach is motivated by two intuitions: 1. Look-ahead code contains fewer dependences and thus lends itself to (speculative) parallelization; and 2. Without correctness constraints, hardware support for speculative parallelization of the look-ahead thread can be much less demanding. We have presented a software mechanism to probabilistically extract parallelism and shown that indeed the look-ahead code affords more opportunities. We have also presented a hardware design that does not contain the array of support needed for conventional speculative parallelization such as dependence tracking and complex versioning. For an array of 14 applications where the speed of the look-ahead thread is the bottleneck, the proposed mechanism speeds up the baseline, single-threaded look-ahead system by up to 1.39x with a mean of 1.13x. Experimental data also suggest there is further performance potential to be extracted which would be investigated as future work.

References

[1] 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 *Proc. Int'l Symp. on Comp. Arch.*, pages 484–295, June 2009.

[2] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, , and M. Upton. Continual Flow Pipelines. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, October 2004.

[3] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. Int'l Conf. on Supercomputing*, pages 68–75, July 1997.

[4] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 129–140, February 2003.

[5] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, May/June 2005.

[6] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed Early Load Retirement. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 16–27, February 2005.

[7] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction. *IEEE TCCA Computer Architecture Letters*, 3, December 2004.

[8] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. Int'l Symp. on Comp. Arch.*, pages 14–25, June 2001.

[9] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. Int'l Symp. on Microarch.*, pages 59–68, November–December 1998.

[10] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. Int'l Symp. on Comp. Arch.*, pages 186–195, May 1999.

[11] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proc. Int'l Symp. on Comp. Arch.*, pages 2–13, June 2001.

[12] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proc. Int'l Symp. on Comp. Arch.*, pages 52–61, June 2001.

[13] C. Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 40–51, June 2001.

[14] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 37–48, January 2001.

[15] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice-processors: an Implementation of Operation-Based Prediction. In *Proc. Int'l Conf. on Supercomputing*, pages 321–334, June 2001.

[16] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. In *Proc. Int'l Symp. on Microarch.*, pages 269–280, December 2000.

[17] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu. Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining. In *Proc. Int'l Symp. on Microarch.*, pages 387–398, December 2003.

[18] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 231–242, September 2005.

[19] F. Mesa-Martinez and J. Renau. Effective Optimistic-Checker Tandem Core Design Through Architectural Pruning. In *Proc. Int'l Symp. on Microarch.*, pages 236–248, December 2007.

[20] B. Greskamp and J. Torrellas. Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 213–224, September 2007.

[21] A. Garg and M. Huang. A Performance-Correctness Explicitly Decoupled Architecture. In *Proc. Int'l Symp. on Microarch.*, November 2008.

[22] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 414–425, June 1995.

[23] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proc. Int'l Symp. on Microarch.*, pages 85–96, November 2002.

[24] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. Int'l Symp. on Microarch.*, pages 226–236, November–December 1998.

[25] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 2–13, January–February 1998.

[26] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. Int'l Symp. on Comp. Arch.*, pages 13–24, June 2000.

[27] S. Balakrishnan and G. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. In *Proc. Int'l Symp. on Comp. Arch.*, pages 302–313, June 2006.

[28] P. Xekalakis, N. Ioannou, and M. Cintra. Combining Thread Level Speculation, Helper Threads, and Runahead Execution. In *Proc. Intl. Conf. on Supercomputing*, pages 410–420, June 2009.

[29] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 257–268, November 2000.

[30] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Energy-Efficient Thread-Level Speculation on a CMP. *IEEE Micro*, 26(1):80–91, January/February 2006.

[31] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, pages 195–205, January–February 1998.

[32] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[33] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, September 2000.

[34] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, September 2005.

[35] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proc. Int'l Symp. on Comp. Arch.*, pages 24–33, April 1994.

[36] I. Ganusov and M. Burtscher. On the Importance of Optimizing the Configuration of Stream Prefetchers. In *Proceedings of the 2005 Workshop on Memory System Performance*, pages 54–61, June 2005.

[37] S. Sair and M. Charney. Memory Behavior of the SPEC2000 Benchmark Suite. Technical report, IBM T. J. Watson Research Center, October 2000.