A Performance-Correctness Explicitly-Decoupled Architecture

Alok Garg and Michael C. Huang
Department of Electrical & Computer Engineering
University of Rochester
{*garg, huang*}*@ece.rochester.edu*

Abstract

Optimizing the common case has been an adage in decades of processor design practices. However, as the system complexity and optimization techniques' sophistication have increased substantially, maintaining correctness under all situations, however unlikely, is contributing to the necessity of extra conservatism in all layers of the system design. The mounting process, voltage, and temperature variation concerns further add to the conservatism in setting operating parameters. Excessive conservatism in turn hurt performance and efficiency in the common case. However, much of the system's complexity comes from advanced performance features and may not compromise the whole system's functionality and correctness even if some components are imperfect and introduce occasional errors. We propose to separate performance goals from the correctness goal using an explicitly-decoupled *architecture.*

In this paper, we discuss one such incarnation where an independent core serves as an optimistic *performance enhancement engine that helps accelerate the correctness-guaranteeing core by passing high-quality predictions and performing accurate prefetching. The lack of concern for correctness in the optimistic core allows us to optimize its execution in a more effective fashion than possible in optimizing a monolithic core with correctness requirements. We show that such a decoupled design allows significant optimization benefits and is much less sensitive to conservatism applied in the correctness domain.*

1. Introduction

Achieving high performance is a primary goal of processor microarchitecture design. While designs often target the common case for optimization, they have to be correct under all cases. Consequently, while there are ample opportunities for performance optimization and novel techniques are constantly being invented, their practical application in real product designs faces ever higher barriers and costs, and diminishing effectiveness. Correctness concern, especially in thorny corner cases, can significantly increase design complexity and dominate verification efforts. The reality of microprocessor complexity, its design effort, and costs [1] reduces the appeal of otherwise sound ideas, limits our choice, and forces suboptimal compromises. Furthermore, due to the tightly-coupled nature of monolithic conventional microarchitecture, conservatism or safety margin necessary for each component and each layer of the design stack quickly accumulates and erodes common-case efficacy. With mounting PVT (process, voltage, and temperature) variation concerns [2], the degree and extent of conservatism will only increase. The combination of high cost and low return makes it increasingly difficult to justify implementing a new idea and we need to look for alternative methodology that allows us to truly focus on the common case. One promising option is to explicitly decouple the circuitry for performance and correctness goals, allowing the realization of each aspect to be more efficient and more effective. Decoupling is a classic, time-tested technique and seminal works on various types of decoupling in architecture [3]–[7] have attested its effectiveness and advanced the knowledge base. Building on this foundation, we propose to explore *explicitly-decoupled* architecture (EDA).

By "explicit", we mean two things. First, the decoupling is not simply providing a catch-all mechanism for a monolithic high-performance microarchitecture to address rare-case correctness issues. Rather, from ground up, the design is explicitly separated into a performance and a correctness domain. By design, the performance domain only *enables* and *facilitates* high performance in a probabilistic fashion. Information communicated to the correctness domain is treated as fundamentally speculative. Therefore, correctness issues in the performance domain will only affect its performance-boosting capability and become a performance issue for the whole system. This allows true focus on the common case and reduction of design complexity, which in turn permits the implementation of ideas previously deemed impractical or even incorrect. An effective performance domain allows designers to use simpler, throughput-oriented designs for the correctness domain and focus on other practical considerations such as system integrity.

Second, the architecture design is not just conceptually but also physically partitioned into performance and correctness domains. The physical separation extends to the whole system stack from software and microarchitecture down to circuit and device. Physical separation ① allows the entire system stack to be *optimistically* designed; ② conveniently and economically

This work is supported by NSF CAREER award CCF-0747324 and also in part by grants CNS-0719790 and CNS-0509270.

provides the same mechanism for ultimate correctness guarantee; and ③ permits custom software-hardware interface in the performance domain, which opens up more cross-layer cooperative opportunities to implement ideas difficult to accomplish within a single layer.

Explicitly decoupled architecture represents a very broad design space. What aspect of performance improvement is best achieved in such a decoupled fashion, and how to exploit the lack of concern for correctness to design novel optimization techniques and indeed synergistic techniques from different layers are but a small set of questions that need to be addressed. To narrow down the exploration, in this paper we are focusing on using the EDA principle to improve traditional ILP (instruction-level parallelism) lookahead, and study the effect of using practical, complexity-effective techniques to manage long, and more importantly, unpredictable latencies associated with branch and load processing. The discussed design is by no means a mature final product, but rather a proof of concept that hopefully helps to reveal some insights.

The rest of the paper is organized as follows: Section 2 explains some high-level design decisions; Section 3 discusses the basic support needed to enable an explicitly-decoupled execution; Section 4 discusses several opportunities to achieve complexity-effective performance optimization; Section 5 presents quantitative analyses; Section 6 discusses related work; and Section 7 summarizes and discusses some future work. Due to space constraints, some details are left in the technical report [8].

2. High-Level Design Decisions

While lookahead techniques have the potential to uncover significant amount of ILP, conventional microarchitectures impose practical limitations on its effectiveness due to their monolithic implementation. Correctness requirement limits the design freedom to explore probabilistic mechanisms and makes conventional lookahead resource-intensive: registers and various queue entries need to be reserved for every in-flight instruction, making deep lookahead very expensive to support. Moreover, the design complexity is also high as introduction of any speculation necessitates fastidious planning of contingencies.

In contrast to this "integrated" lookahead design, in an EDA, a decoupled agent is to provide the lookahead effort. Furthermore, we also want to minimize the mutual dependence between the lookahead agent on the normal processing agent (the *optimistic* and the *correctness* core, respectively in our design shown in Figure 1). This decision has implications on how we maintain autonomy of the cores and manage the deviance between them.

Autonomy. A key point of our design is that the optimistic core can be *specialized* to perform lookahead more effectively by leveraging the lack of correctness constraints. To maintain autonomy of the lookahead with respect to normal processing, we use an independent thread of control – the optimistic thread. Having its own thread of control in the optimistic core also allows us to freely exploit speculative, optimistic software analyses or transformations.

We could simply use another copy of original program binary as the optimistic thread. This is straightforward but suboptimal. A skeletal version of the program that contains only instructions relevant to future control flow and data accesses is enough. There is no need to include computation that is only necessary for producing the right program output and non-essential for lookahead. We can rely on software analysis to generate such a "skeleton" in a probabilistic fashion. In this paper, our software analysis is done on the program's binary. Performing the tasks on binaries has the significant benefit of hiding all the implementation details beneath the contractual interface between the hardware and the programs, and maintaining semantic binary compatibility: In each *incarnation* of an EDA, we can customize the instruction set of the optimistic core without worrying about future compatibility obligations.

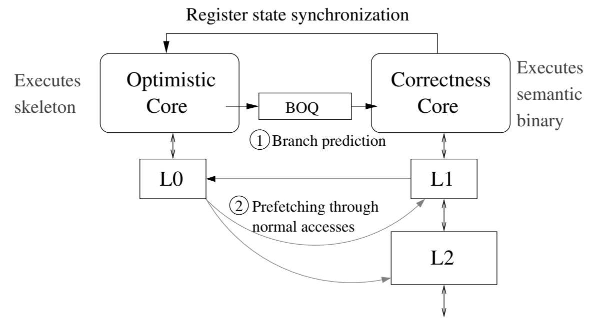


Figure 1. The optimistic core, the correctness core, and the organization of the memory hierarchy. The optimistic core ① explicitly sends branch predictions to the correctness core via the branch outcome queue (BOQ) and ② naturally performs prefetching with its own memory accesses.

Managing deviance. The removal of correctness constraints in the performance domain provides the freedom to explore cost-effective performance-boosting mechanisms and avoid excessive conservativeness. However, it would inevitably lead to deviation from the desired result. For example, approximations in the skeleton generation, a logic simplification in the architecture design, or device glitches due to insufficient margin can all cause the architectural state in the performance domain to deviate from the desired state. If the design heavily depends on vast amounts of predictions and on the preciseness of the predicted information from the performance domain, such deviations are likely to result in costly remedies in the correctness domain and ultimately limit our freedom in exploring unconventional and optimistic techniques.

To build in an inherent tolerance for such deviations, we do not rely on the optimistic core to provide value predictions and only draw branch direction predictions from it. This is done using a FIFO structure *Branch Outcome Queue* (BOQ) as shown in Figure 1. This also allows us to detect the control flow divergence between the two threads. When this happens, the correlation between the execution of the two threads is reduced and at some point, the state of the optimistic core needs to be reinitialized to maintain its relevance in lookahead. We call this

a *recovery*. In this paper, for simplicity, a recovery is triggered whenever a branch misprediction is detected in the correctness core and a recovery involves copying architectural register state from the correctness core to the optimistic core. We note that while we are actively exploring alternatives, we have not found a design with superior performance.

Even with the recovery mechanism, memory writes in the performance domain are still fundamentally speculative and need to be contained within its local cache hierarchy. We use one private cache (L0, for notional convenience). By simply sharing the rest of the memory hierarchy between the two cores, we can tap into the rest of the architectural state in a complexity-effective manner. L0 never writes back anything to the rest of the hierarchy.

Related work. While we will discuss related work in more detail later, it is worth highlighting here the key differences. Note that the differences are often the result of difference in goal.

Decoupling correctness and performance issues is not a new concept. We want to make a case for a more explicit, up-front approach to decoupling, which makes performance optimization and correctness guarantee more independent than prior art. This is reflected in

1) The division of labor in the two cores: The optimistic core is only attempting to facilitate high performance by passing hints and other meta data. In the common case, it only provides good hints, whereas the leading cores in [3], [7], [9] will produce complete and correct results.
2) The minimal mutual dependence between them: Neither does the trailing core require a large amount of accurate information from the leading core (such as architectural state to jump start future execution [5]), nor does the leading core heavily depend on the trailing core to perform its task [6].

3. Basic Support

The potential of explicitly-decoupled architecture lies in the opportunities it opens up for *efficient* and *effective* optimizations. The required support to allow the optimistic core to perform self-sustained lookahead is rather basic and limited.

3.1. Software Support

A key requirement for the envisioned system to work effectively is that the optimistic core has to *sustain* a performance advantage over the correctness core so as to allow *deep* lookahead. A key opportunity is that the skeleton only needs to perform proper data accessing, which is only part of the program, and may be able to skip the remainder. This is not a new concept. Indeed, the classic access/execute decoupled architecture [6] exploits the same principle to allow the access stream to stay ahead. However, the challenge is that our optimistic core is significantly more independent and has to do enough work to ensure a highly accurate control flow. As it turns out, using conventional analysis on the binary, we can not successfully remove a sufficient number of instructions: about 10-12% dynamic instructions (most of which prefetches) can be removed from the program binary without affecting the program control flow. This is not sufficient to sustain a speed advantage for the optimistic thread. While extremely biased branches (identified through profiling) can be removed or turned into unconditional branches reducing the need for some branch condition computation, solely relying on this is also insufficient.

A simple but important observation is that the optimistic thread has access to the architectural memory hierarchy in the correctness domain and therefore can obtain the data from memory when the producer store is sufficiently upstream in the instruction sequence that at the time of load – it would have been executed by the correctness core. We do not need to include the store and its backward slice in the skeleton. Note that this is also exploited earlier in [5].

3.2. Architectural Support

The architectural support required to enable our explicitly decoupled architecture is also limited. For the most part, both cores operate as self-sufficient, stand-alone entities. The only relatively significant coupling between the two cores is that the correctness core's memory hierarchy also serves as the lower levels of the memory hierarchy for the optimistic core. Note that, the accesses from the optimistic core to L1 is infrequent as it only happens when the L0 misses. Hence, extra traffic due to servicing L0 misses is insignificant. Indeed, as we will show quantitatively later, the increase in L1 traffic is more than offset by the decrease of L1 accesses from the correctness core because of better branch prediction.

Another support needed is the recovery mechanism. A branch outcome queue (BOQ) is used to pass on the branch direction information from the optimistic core to the correctness core. When such a prediction is detected in the correctness core as incorrect, a recovery is triggered. The correctness core drains the pipeline and passes the architectural register state back to the optimistic core. Since the L0 cache is corrupted because of wrong-path execution, some cleansing may be helpful. For simplicity, we reset the entire L0 cache upon a recovery. Also, the fetch stage of the correctness core is frozen when the BOQ is empty. This ensures the "alignment" of the branches: the next branch outcome to be deposited by the optimistic core in the BOQ is always intended for the next branch encountered by the correctness core.

4. Opportunities

By separating out correctness concerns, EDA allows designers to make different trade-offs and devise more effective performance optimization strategies. A primary implication of decoupling is that not all mis-speculations need to be corrected or even detected in the performance domain. In a conventional design that tightly couples correctness and performance, the complexity of such detection and recovery logic may significantly affect cost-effectiveness of the implementation, reduce the appeal of an otherwise sound idea, and can even defeat the purpose of

speculation. In EDA, designs can use new, probabilistic mechanisms to explore optimization opportunities in a more cost-effective way and avoid the complex algorithms and circuitry that place stringent requirements on implementation. We discuss a few opportunities that we have explored.

4.1. Skeleton Construction

Recall that the skeleton does not need to contain long-distance stores and their computation chain. However, the communication relationship between loads and stores is not always clear, especially when dealing only with program binaries. Fortunately, our binary parsing only needs to approach the analysis in a probabilistic fashion, and we can use profiling to easily obtain a statistical picture of load-store communication patterns. The process is as follows and we use a binary parser based on `alto` [10] to perform the analysis and transformations.

Profiling. We first perform a profiling step to obtain certain information parsing the binary alone can not. First of all, we can obtain the destinations of indirect jump instructions. Again, we do not need to capture all possible destinations, thanks to the lack of correctness requirement for the optimistic core. With this information, we can make the control flow graph more complete.

Secondly, we collect statistics about short-distance load-store communications. Using a training input, we obtain the list of stores with *short instances*. A short instance is a dynamic store instance whose consumer load is less than d_{th} instructions downstream. We set d_{th} to 5000 in this paper. We found that the profile results are not sensitive to d_{th}. For every store with short instances, we tally the total number of dynamic instances as well as short instances. For the latter, the statistics are further subdivided based on the identity of their consumer loads. This is needed in later analysis because whether a short instance matters depends on if the consumer load is part of the skeleton.

Finally, the profiling run also identifies load instructions that are likely to miss in the (L2) cache and branches with strong biases. The statistical miss frequencies are recorded for later analysis. Branch bias factored with cost (additional instructions added) is used to label some branches as biased. In general, these branches have a bias greater than 99.9%.

Binary analysis. With this profile information, we then proceed to build a program skeleton. The goal of the skeleton is to closely track the original program's control flow and be able to pass on branch prediction information and issue timely prefetches. Thus, the first thing we do is to mark branch instructions as selected in the skeleton. Next, traversing the data-flow graph backward, we mark all the instructions on the backward slice of the branch instructions. Branches considered extremely biased are turned into NOPs or unconditional branches and therefore they do not have any backward slice. Following this, we need to deal with memory dependences and include producer stores that feed into the loads included in the current skeleton. Our goal is to minimize the total computation due to included stores and at the same time keep the total number of short instances from excluded stores small. The aforementioned profile information about short instance helps us to determine which stores to keep. We ignore short instances involving a load not included in the skeleton and sort stores with increasing ratio of short to total instances. We walk down the list and exclude the top ranking stores until the total short instances from them surpasses 10,000th of total dynamic instruction count. Since trimming stores from the skeleton changes which loads belong to the skeleton and affect the ratio for ranking, we iterate the analysis a few times for a better result.

Finally, we insert prefetch instructions for those loads likely to miss in the cache and are not already included in the skeleton. Whether to include a particular load is also determined by its cost-benefit ratio. The benefit (of adding a prefetch) is approximated as the miss penalty multiplied by the miss probability. The cost is approximated by the number of instructions added to compute the address. If the ratio is lower than a threshold (empirically set to 3), the prefetch is inserted.

Eliminating useless branches. Note that in terms of what information to pass between the two domains in an EDA and how to obtain that information in the performance domain, the design space is vast. The basic skeleton we formed is a code that not only strives to stay on the right path to maintain relevance, but also attempts to execute *every* branch in the original semantic binary. This is a design choice, not a necessity to support deep lookahead. We explore this option because handling frequent branch misprediction is a necessity that affects all microarchitectural components. If the correctness domain can expect a highly-accurate stream of branch predictions, its microarchitecture can be fundamentally simplified. Because of this choice, we found that the skeleton includes branches completely useless for its own execution. These include empty if-then-else structures and sometimes empty loops as shown in Figure 2. In these cases, including the branch can be very inefficient, especially in the case of empty loops: when the loop branch is biased and turned into an unconditional branch, the optimistic thread will be "trapped" in the loop until the trailing correctness thread catches up, finishes the same loop, and generates a recovery. Not only will the optimistic thread forfeit any lead upon reaching the empty loop, it also wastes energy from then on until recovery doing absolutely nothing useful.

In these cases, by not executing the branch, we avoid unnecessary waste in the optimistic core and may even manage to avoid a costly recovery. It is straightforward to identify these branches using the parser. The only issue when skipping them is that of branch "alignment": Because there is a one-to-one correspondence of branches between the optimistic and correctness thread (so as to use a simple FIFO for the BOQ), if the optimistic thread skips a branch, the correctness thread will (mis)interpret the next piece of prediction as that of the skipped branch, thus losing alignment.

We maintain alignment by replacing the branch to be skipped with a special branch instruction in the skeleton. Specifically, we add three types of branches: BDC, BUT, and BUF as discussed in Table 1.

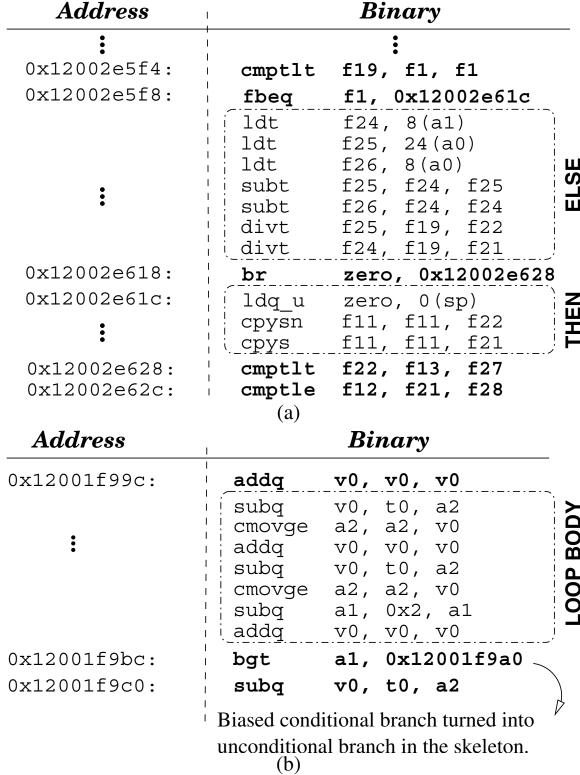


Figure 2. Examples of empty if-then-else block (a) and loop (b) in the skeleton of real applications. Instructions selected in the skeleton are shown in bold.

Symbol	Correctness thread interpretation and action
BDC (don't-care)	Branch prediction unavailable for this branch.
BUF (until fall-thru)	Branch prediction unavailable for the loop. Stop drawing from BOQ until this branch falls through.
BUT (until taken)	Branch prediction unavailable for the loop. Stop drawing from BOQ until this branch is taken.

Table 1. Replacing useless branches in the skeleton.

4.2. Cost-Effective Architectural Support

Due to space constraints, out of the several architectural mechanisms we studied we only discuss a few, with a particular emphasis on simplicity of the designs as in practice, complicated techniques tend to be avoided by real-world designers.

Simplistic value substitution. Stalling induced by off-chip accesses can seriously impact the optimistic thread. Given the freedom we enjoy in the optimistic core, there are quite a few options to avoid waiting for an off-chip memory access. Perhaps the simplest (and indeed a simplistic) way is to give up waiting and feed some arbitrary value to the load instruction in order to *naturally* flush it out of the pipeline. This may seem senseless as a wrong value may cause the optimistic thread to veer off the correct control flow and render it irrelevant and maybe even harmful. However, there are several natural tolerance mechanisms that come to the rescue: the data loaded may not be control-flow related but is part of the prefetching effort; the load may even be dynamically dead; and the error in the value may be masked by further computation or comparisons. We show two examples of masking from real benchmarks.

Figure 3-(a) shows a very typical code sequence where the loaded value is compared to a constant to determine branch direction. Figure 3-(b) shows another kind of masking. In this example, under certain conditions, the loaded value is canceled out in the computation and no longer matters. In summary, in many cases, the exact value of a load does not matter and it is more important to flush the long-latency instruction out of the system so as to continue useful work downstream rather than to wait for the memory to respond – unless the optimistic thread is sufficiently ahead. We only feed a substitute value when the distance between the two threads is below a threshold. This distance is measured by the number of branch predictions in the BOQ.

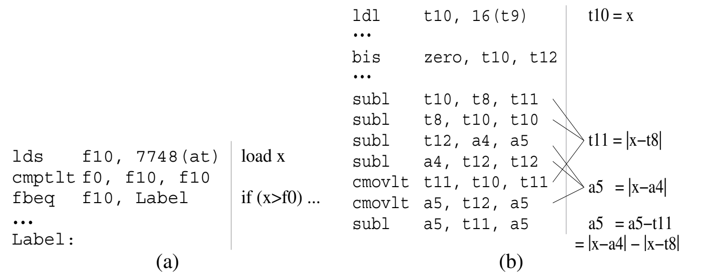


Figure 3. The inherent masking effect in real programs.

In terms of determining the substitute value, we can obviously use a conventional value predictor or even a special-purpose predictor [11]. However, we opt for the much simpler approach of providing a 0, partly because it is the most frequently occurring value in general.

We note that an apparent alternative is to explicitly flush out the dependence chain of the load instruction as done in [12]. The primary benefit of our approach is its simplicity: An independent logic determines when to use value substitution and when it is used, the rest of the core is unchanged – there is no extra logic to explicitly tag results as invalid and propagate the "poison". Secondarily, as our examples show, in some cases, the exact value may not matter much. Explicitly flushing the apparent dependence chain without executing them prevents the prefetching benefit.

Clearly, zero value substitution does not always work well. In particular, when the value is some form of an address, substituting a zero is often a sure way to get into trouble. A light-weight solution we adopted is to identify "address" loads using the parser and encode them differently to prevent the hardware from doing zero value substitution. In other words, these loads will stall if they miss in the L2. We choose this because the hardware support needed is minimum.

Delayed release of prefetches. If the optimistic thread achieves very deep lookahead, we do not want the prefetches to be issued too early. Thus we record the addresses into a *prefetch address buffer* (PAB) together with a timestamp indicating the appropriate future moment to release it. This time is set to be

about one memory access latency prior to estimated execution time of the load in the correctness thread. One subtle issue we encountered is the timing of address translation. Since loading with a virtual address can cause a TLB miss, which can potentially take another off-chip access, we try to put translated address into the PAB. However, if the translation causes a TLB miss, instead of blocking and waiting for the TLB to fill, we keep the virtual address and perform the translation when the prefetch is released from the PAB.

Managing stale data. As discussed above, the optimistic core is relying on the correctness core to provide the data when the distance between a load and its producer store is long. To allow the data to be obtained from the correctness core, *i.e.*, from the L1 cache or beyond, stale data should be removed from the L0 cache to force an access to the L1. Fortunately, the cache's replacement algorithm, which typically uses LRU-like policies, already purges some stale data out of the cache naturally. We also explored other proactive options, including turning the store into invalidations and adding a timer-based eviction mechanism. None of these approaches brings enough consistent benefits (1-2% performance gain in our simulations) to justify the extra complexity. In our final design, we leave it to the cache replacement to probabilistically evict undesired cache lines. We couple that with periodically forced recoveries: if no recovery has occurred for a long time (150,000 cycles in our experiments), we force the optimistic core to synchronize with the correctness core, cleaning the registers and the L0.

4.3. Complexity Reduction

One important advantage of using EDA is the possibility to significantly reduce the circuit and design complexity of microarchitectural structures. In the optimistic core, we can afford to focus only on the common case and have designs that do not always work. In the correctness core, we can use simpler algorithms and less ambitious implementations as there is less need to *aggressively* exploit ILP. It can use a simpler, throughput-optimized microarchitecture and smaller, less power-hungry structures. Because the core bears the burden of guaranteeing the correctness, a much simpler implementation can have a series of benefits. For instance, having fewer timing critical paths means that the whole core is less vulnerable to PVT variation concerns. Using a simpler and smaller core also makes fault tolerance easier and more efficient. Here we discuss a few straightforward opportunities to reduce complexity from a generic microarchitecture to serve as cores inside an EDA. We leave the exploration of special-purpose throughput-oriented design as future work.

Correctness core. Perhaps the most important simplification opportunity comes from branch handling in the correctness core, thanks to the much more accurate branch directions provided by the optimistic core. Conventional architecture requires immediate reaction upon a detected misprediction and needs the capability to (a) quickly restore register alias table (RAT) mapping; and (b) purge *only* wrong-path instructions and their state from various microarchitectural structures such as the issue queue, the LSQ, and the re-order buffer. In contrast, because mispredictions are truly rare (see Section 5), the correctness core does not need instant reaction. Instead, upon the detection of a misprediction, the core can drain the right-path instructions and reset the pipeline – no partial repair is needed. Hence, checkpointing of RAT upon branch instruction is no longer necessary. This has a secondary effect of reducing the possibility of stalling when running out of RAT checkpoints.

Additionally, the characteristics of the program execution in the correctness core is different from that in a conventional core. First, cache misses are significantly mitigated. Thus, the core will be less sensitive to the reduction of in-flight instruction capacity. Second, with the optimistic core taking care of lookahead, latency of various operations becomes less critical so long as the throughput is sufficient. For example, the system's overall performance will be less sensitive to modest frequency changes in the correctness core. This makes it easy to use conservatism to deal with variations. Third, advanced features in the microarchitecture can be avoided. For example, load-hit prediction is widely used in order to schedule dependents of loads as early as possible [13]. The result of such speculation is extra complexity dealing with mis-speculation and supporting *scheduling replays* [14]. We can do away with such speculation in the correctness core. Another example is to simplify the issue logic with some form of in-order constraints, such as in-order issue *within* any reservation station/queue. This would eliminate the circuitry to compact empty entries and simplify the wakeup-select loop.

In short, a whole array of complexity-performance tradeoffs can be performed. As we do not have quantitative models of the complexity benefit in terms of design effort reduction or critical path length reduction, we show the performance sensitivity of these complexity reduction measures in Section 5. In particular, we show that the performance impact of a simplification is in general much lower than in a conventional monolithic microarchitecture.

Optimistic core. Logic blocks in the optimistic core can also be simplified. Such a simplification can even trade off correctness for lower complexity and better circuit timing. Take the complex memory dependence logic for example. We first eliminate the LQ altogether, ignoring any potential replays due to memory dependency or coherence/consistency violation. We also simplify forwarding in the SQ by removing the priority encoding logic, which is considered a scalability bottleneck of SQ. Rather than relying on the priority encoder to select the right store among multiple candidates for forwarding, we ensure that at most one store (the youngest) will respond to a search. This is achieved by disabling the entry of an older store with the same address upon the issue of a new store [8]. Finally, we bypass TLB access and use virtual address to search the SQ, ignoring virtual address aliasing.

5. Experimental Analysis

5.1. Experimental Setup

Simulator support. We perform our experiments using an extensively modified version of SimpleScalar [15]. Support was added to allow modeling of EDA and of value-driven simulation. We made extensive modifications to increase fidelity in the memory subsystem and microarchitecture[1] and to support leakage and dynamic power analysis. For brevity, the details are left in [8].

	Baseline core	Aggressive core
Fetch/Decode/Commit	8 / 4 / 6	16 / 6 / 12
ROB	128	512
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div	INT 3+1 mul +1 div, FP 3+1 mul +1 div
Issue Q / Reg. (int,fp)	(32, 32) / (80, 80)	(64, 64) / (400, 400)
LSQ(LQ,SQ)	64 (32,32) 2 search ports	128 (64,64) 2 search ports
Branch predictor	Bimodal + Gshare	Bimodal + Gshare
- Gshare	8K entries, 13 bit history	1M entries, 20 bit history
- Bimodal/Meta/BTB	4K/8K/4K (4-way) entries	1M/1M/64K (4-way) entries
Br. mispred. penalty	at least 7 cycles	at least 7 cycles
L1 data cache	32KB, 4-way, 64B line, 2 cycles, 2 ports	48KB, 6-way, 64B line, 2 cycles, 3 ports
L1 I cache (not shared)	64KB, 1-way, 128B, 2 cyc	128KB, 2-way, 128B, 2 cyc
L2 cache (uni. shared)	1MB, 8-way, 128B, 15 cyc	1MB, 8-way, 128B, 15cyc
Memory access latency	400 cycles	400 cycles
Correctness core:	Baseline core without branch predictor and with circuit simplifications discussed in Section 4.3.	
Optimistic core:	Baseline core with microarchitectural design discussed in Section 4.2 L0 cache: (16KB, 4-way, 32B line, 2 cycle, 2 ports). Round trip latency to L1 is 6 cycles	
Communication:	BOQ: 512 entries; PAB: 256 entries; register copy latency (during recovery): 32 cycles	
Process specifications:	Feature Size: 45nm; Frequency: 3 GHz; V_{dd}: 1 V	

Table 2. System configuration.

Applications, inputs, and architectural configurations. We use highly-optimized Alpha binaries of SPEC CPU2000 benchmarks. For profiling, we use the *train* input and run the applications to completion. For evaluation, we simulate 100 million instructions after skiping over the initialization portion as indicated in [16] using *ref* input. Our baseline core configuration, is a generic high-end microarchitecture loosely modeled after POWER4's core [17]. To provide a reference for the lookahead effect of EDA, we also use a very aggressively configured core. Details of the configurations are shown in Table 2.

5.2. Benefit Analysis

A primary benefit of EDA is that it allows *complexity-effective* designs – by using simple mechanisms but targeting high-impact performance bottlenecks. EDA allows design effort to be focused more on exploring new opportunities rather than on ensuring an optimization technique actually works in silicon all the time. Unfortunately, we are not yet capable of quantifying design effort nor can our current design – far from mature – be used to convincingly justify the upfront cost of explicit separation and the resulting partial redundancy.

In the following we hope to offer some evidence that this paradigm is worth further exploration. A particular point we want to emphasize is the no single aspect (*e.g.*, absolute performance) taken in isolation can be construed as a figure of merit. Put together, these results show that even with only intuitive and straightforward techniques, the discussed design still (a) achieves good performance boosting, (b) does not consume excessive energy, and (c) provides robust performance and better tolerance than conventional design to circuit-level issues and to the resulting conservatism.

Performance gain of optimism. Figure 4 shows the speedup of the proposed EDA over a baseline core and the speedup of expanding the baseline core into an impractically aggressive monolithic core. Since our EDA is performing traditional ILP lookahead, it is not surprising to see the two options follow the same trend: in general, floating-point codes tend to benefit more noticeably. On average, EDA's speedup is more pronounced. The geometric means are 1.49 (INT) and 1.96 (FP) compared to the aggressive core's 1.29 (INT) and 1.49 (FP).

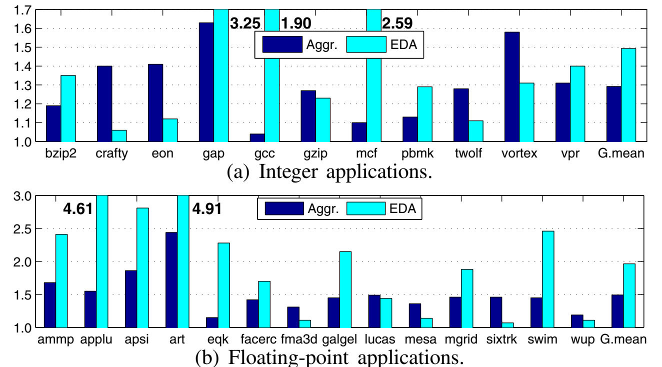


Figure 4. Speedup of proposed EDA and an aggressively configured monolithic core (Aggr.) over baseline for SPEC INT (a) and FP (b). Detailed IPC results are left in [8].

While we are focusing on ILP exploitation, the effect can be equally important in explicitly parallel programs. We have applied the same methodology to parallel programs – ignoring any opportunity to target shared-memory issues in the design of EDA – and have also observed significant performance gains. Figure 5 summarizes the performance gain of SPLASH applications running on a CMP with EDA cores compared to that with conventional cores. As a reference, a configuration with twice as many cores (16-way) is also shown. An important point to note is that exploiting ILP is not guaranteed to be less effective than exploiting TLP (thread-level parallelism) for parallel codes. As can be seen, even for these highly-tuned scientific applications, scaling to more cores does not give perfectly linear speedup and in some cases producing (far) lower return than improving ILP. Exploring multiprocessor-aware optimistic techniques in EDA will likely unleash even more performance potential.

The results suggest that decoupled lookahead can uncover some parallelism not already exploited in a state-of-the-art microarchitecture, at least for some applications. In some cases, the return is significant. Note that the specific numbers are secondary as the design point is chosen to illustrate the potential rather than to showcase an optimized design. Indeed, as we show

1. One such modification significantly changes baseline performance. Without this, the benefit of our design – and decoupled architectures in general – will be exaggerated.

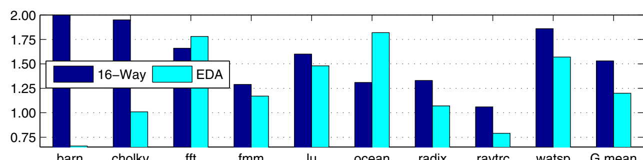


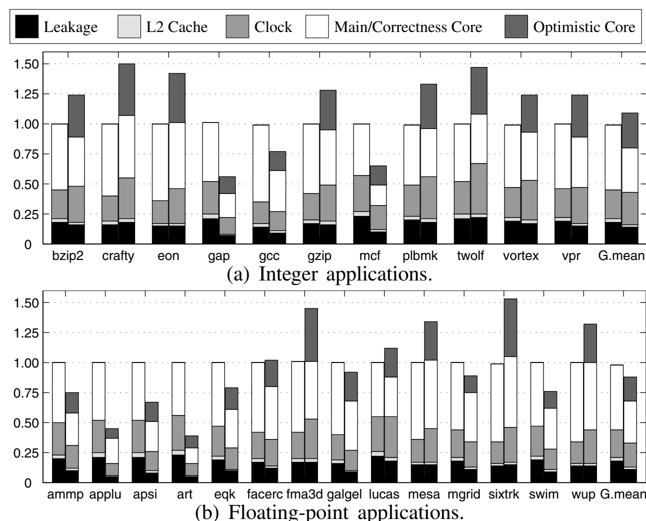
Figure 5. Speedup of SPLASH applications running on an 8-way CMP. Each conventional core in CMP system is replaced by an EDA core. For contrast, the speedup of a 16-way CMP is also shown. In some cases, enhancing ILP even outperforms doubling the number of cores.

later, the correctness core can be much less aggressive with virtually no performance impact. The key point is that this is achieved with simple techniques designed to minimize circuit-implementation challenges. Given the correctness decoupling, the entry barrier for implementing other optimizations in the performance domain should be considerably lower than in a conventional system. More investigation would discover other "low-hanging fruits" to achieve high performance with low complexity. In turn, the performance "currency" can be used to pay for other important design goals.

Energy implications. An apparent disadvantage of an EDA is that it is power-inefficient: roughly two cores are used and the program needs to execute twice and this may lead one to believe that energy consumption would double. In reality, the energy overhead can be far less due to a number of factors. First, the skeleton is after all not the entire program and in certain applications can be quite small (Section 5.3). Second, many energy components do not double. For instance, only the optimistic core wastes energy executing wrong-path instructions following mispredicted branches. Third, when a prefetch is successful, it is a good energy tradeoff to execute a few more instructions to avoid a long-latency stall which burns power while doing nothing. As can be seen in Figure 6, which shows normalized energy consumption with breakdown in both EDA and baseline systems, a significant performance improvement results in lower energy consumption in EDA. Indeed, EDA results in an average of 11% energy *reduction* for FP applications. Even for integer codes, the energy overhead of EDA is only 10%.

While there is clearly room for improvement, we note that the study is conservative in that the EDA configuration is hardly optimized for energy-efficiency. For instance, no attempt is made to shut down the optimistic core when it is not effective and no energy benefits of the architectural simplifications (Section 4.3) are taken into account. Moreover, aggressive assumptions are made to reduce residual energy consumption during idling [8].

Performance cost of conservatism in EDA. Unlike the optimistic core where timing glitches or logic errors do not pose a threat to system integrity, the correctness core faces the challenge that it has to be logically correct and functioning reliably despite adverse runtime conditions such as unpredictable PVT variations. Passive conservatism is perhaps still a most practical and effective approach to dealing with the challenge: avoid complexity in the logic design, mitigate timing critical paths, and build in operating margins. The opportunity in an EDA is that



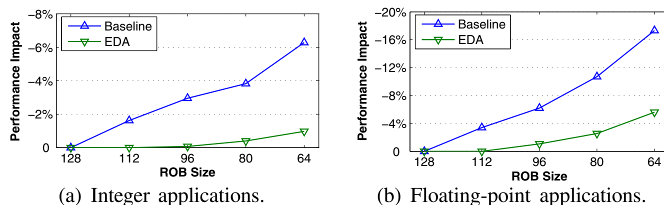
(a) Integer applications.

(b) Floating-point applications.

Figure 6. Normalized energy consumption of EDA (right bar) and baseline systems (left bar). Each bar is further broken down into 5 different sub-categories.

these acts of conservatism do not carry as high a (performance) price tag as in a conventional system. Below, we show that indeed, when we increase the conservativeness in the design and configuration of the optimistic core, the performance impact is insignificant and much less pronounced than doing so in a conventional monolithic design.

First, we hypothesized earlier that the correctness core will be less sensitive to in-flight instruction capacity as it does not need to rely on aggressive ILP exploitation. We study this by gradually scaling down the microarchitectural resources. Figure 7 shows the average performance impact. We scale other resources (issue queue, LSQ, and renaming registers) proportionally with the size of ROB. We contrast this sensitivity with that of a conventional system. The figure clearly shows a much slower rising curve for EDA indicating less performance degradation due to reduction of microarchitecture "depth".



(a) Integer applications.

(b) Floating-point applications.

Figure 7. Performance impact on Baseline and EDA system with reduction in in-flight instruction capacity.

Second, the microarchitectural design of the correctness core can be simplified by avoiding complex performance features. We discussed eliminating load-hit speculation and the necessary scheduling replay support in Section 4.3. Figure 8-(a) shows the performance impact in EDA and in a conventional core. Clearly, load-hit speculation is a useful technique in a conventional core. With decoupled lookahead, its utility in the correctness core is largely negligible – in fact, on average, we even achieve a slight performance improvement for floating-point applications.

This is mainly because with effective lookahead, in floating-point applications, there are abundant ready instructions in the correctness core. Therefore, there is more potential cost and little benefit in doing load-hit speculation. Figure 8-(b) shows the impact of simplifying the integer issue queue to be in-order. Similarly, this is creating a smaller performance penalty than would be in a conventional core. Again, this is because the throughput for integer instructions is generally sufficient and the resulting serialization is less costly than in a conventional core.

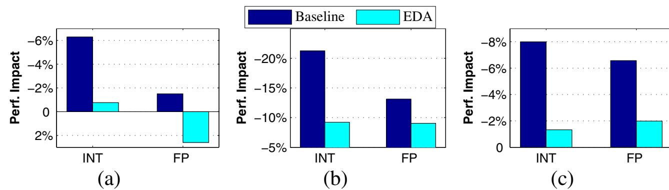


Figure 8. Performance impact of architectural simplification – removing load-hit speculation mechanism (a) and in-order int issue queue (b) and modest clock frequency reduction (c) – in Baseline (BL) and EDA systems.

Third, building in extra timing margin is a simple way to increase a chip's reliability under runtime variations. However, in a conventional system, such margin directly impacts performance. In an EDA, building in timing margin for the correctness core has less direct impact thanks to decoupling. To demonstrate this, we reduce the frequency of the correctness core by 10%. We show the performance degradation and contrast that to the conventional system under the same frequency reduction. From Figure 8-(c), we see that the EDA system has a much smaller sensitivity to such modest frequency reduction. Performance degradation is less than 2% on average, 4-5 times smaller than that of a conventional system.

In summary, as expected, the overall EDA's performance is insensitive to the simplification or extra conservatism introduced to the correctness core – so long as its throughput is sufficient. This means that performance gained using complexity-effective optimistic techniques can be spent to reduce design effort and improve system integrity.

Sensitivity of performance domain circuit errors. In our EDA, the correctness core is not merely providing a correctness safety net and we do not compare the entire output from the two cores and resynchronize whenever there is a difference as some other designs with similar lead/trailing cores such as DIVA [3], Tandem [18], and Paceline [19]. In our current implementation, we only resynchronize when the leading thread veers off the right control flow. (In a future incarnation, even this could be relaxed.) This difference allows the optimistic core to be even less affected by circuit errors (such as due to insufficient margins) than the lead cores in these related designs. This can be shown in a very limited experiment by systematically injecting errors into the committed results and observe their impact on our EDA.

Our injection is different in two respects from common practice. First, we inject multi-bit errors. Unlike particle-induced errors which are found to cause mostly single-bit errors – largely due to limited energy a particle transfers to the silicon, a timing margin failure can cause multi-bit errors. Second, we only inject errors into committed results since we are only interested in finding out the masking effect of our EDA execution model. We use systematic sampling and flip all bits of the result of a committed instruction. Our simulator, which tracks value in all microarchitectural components, allows us to faithfully track the propagation of errors. Figure 9 shows the average number of recoveries incurred per injection at different injection frequencies.

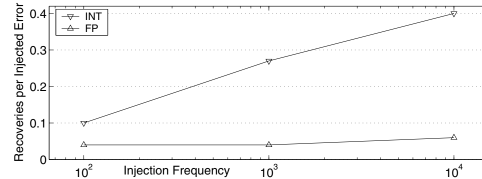


Figure 9. Recoveries per injected error as a function of error injection frequency (number of committed instructions per injected error).

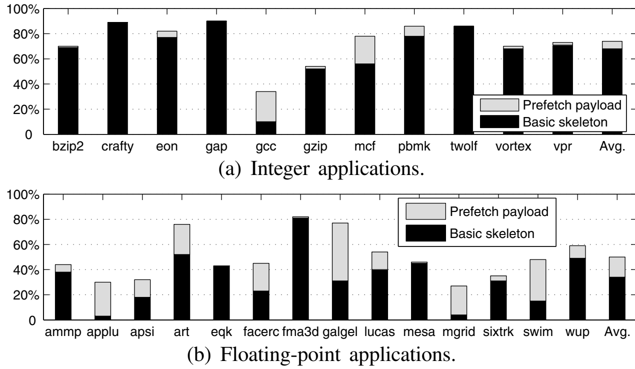
We see that the number varies with the application and in some cases, the lookahead activities can intrinsically tolerate a large degree of circuit operation imperfection. For example, for floating-point applications, out of 20 injected errors, only 1 results in a recovery. Even when circuit errors happen once every 1000 instructions, recovery due to circuit errors will be insignificant. For integer applications, the rate depends on error injection frequency: as error frequency increases, the chance of an error causing a recovery reduces. Intuitively, this is because of increasing likelihood of a single recovery fixing multiple latent errors.

5.3. System Diagnosis

Next, we discuss detailed statistics to help understand the behavior of the system.

Skeleton. We first look at the skeleton generated by the parser. Figure 10 shows the size of the skeleton as the percentage of dynamic instructions in the original binary excluding the NOPs. As would be expected, the skeleton of the integer applications are bigger due to the more complex control flow, whereas the skeleton for floating-point applications are much leaner. On average, excluding prefetches and the special branches (shown in Table 1), the skeleton contains an average of 68% (INT) and 34% (FP) of the instructions from the original binary.

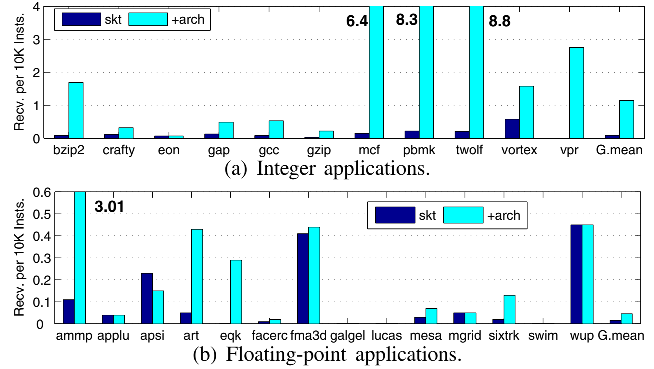
Controlled by a separate thread, the lookahead effort on the optimistic core is no longer tightly bound to (and slowed down by) the actual computation the program has to perform. As seen in Figure 10, the skeleton is quite a bit shorter than the original binary. For some applications, the skeleton also becomes less memory-bound and thus stalled less often when executing. To put the code size reduction into perspective, if we use the conventional baseline microarchitecture to serve as the optimistic core, the effect of the skeletal execution alone can achieve speedup of 0.99 to 2.93 with a geometric mean of 1.27



(a) Integer applications.

(b) Floating-point applications.

Figure 10. Percentage of dynamic instructions left in the skeleton.



(a) Integer applications.

(b) Floating-point applications.

Figure 11. The number of recoveries per 10,000 committed correctness thread instructions. "skt" shows the number due to skeletal execution and "+arch" shows the result after enabling architectural changes in the optimistic core.

in integer codes and 1.04 to 4.51 with a geometric mean of 1.80 in floating-point codes.

Architectural techniques. Within the performance domain, one can employ optimization mechanisms without complex backups to handle mis-speculation or other contingencies. Zero value substitution discussed in Section 4.2 is one example of low-complexity techniques possible in EDA. This simple change resulted in a modest increase in the number of recoveries (about 0.54 per 10,000 committed instructions in integer programs) but allows a net performance benefit of about 13%.

In addition to adding mechanisms for performance gain, we can also take away implementation complexity. For example, in the optimistic core, we drastically simplified the LSQ logic at the expense of correctness (recall that the LQ is completely removed and the priority logic in the SQ is also removed). On average, in 10,000 instructions, less than 0.18 loads receive incorrect forwarding, adding less than 0.05 recoveries. The overall performance cost is virtually negligible (included in results shown in Figure 4). Figure 11 shows the recovery rate due to software and hardware approximations. Note that even in the extreme cases of *mcf*, *perlbmk*, and *twolf*, the benefit of lookahead still outweighs the overhead of recoveries. Also note that while the architectural behavior and the overhead of recoveries are faithfully modeled, the potential positive impact on cycle time or load latency is not considered in our analysis.

Lookahead effects. When the optimistic thread can sustain deep lookahead, it is not difficult to understand why the correctness core is sped up. In our system, both branch mispredictions and cache misses are significantly mitigated in the trailing correctness core. On average, the reduction in misprediction is 90% and 89% for integer and floating-point applications respectively. More analysis about lookahead effect is left in [8].

Finally, we note that backing L0 cache with L1 does not increase the burden of the L1 cache. In fact, with the exception of only a few applications, the total traffic to L1 is reduced, thanks to the reduction in wrong-path instructions in the correctness core. The traffic reduction averages 22%(INT) and 13% (FP) and can be as much as 53%. Accesses from L0 only account for an average of 6% (INT) and 7% (FP) of the total L1 accesses.

Comparison with DCE. If we reduce EDA to a purely performance enhancing mechanism, it resembles a class of techniques represented by decoupled access/execute architecture [6], Slip-stream [7], [20], dual-core execution (DCE) [9], Flea-flicker [21], Tandem [18], and Paceline [19]. Among these, we compare to DCE as architecturally, it is perhaps the most closely related design: both try to avoid long-latency cache miss-induced stalls to improve performance.

Before discussing the statistics, we note that a key design difference stems from the different design goals and the resulting constraints. DCE (and to varying extents, Slip-stream, Flea-flicker, Tandem, and Paceline) focuses on *peripherally augmenting* an existing multi-core microarchitecture. EDA emphasizes on *changing* conventional design practice and making microarchitecture more decoupled (to reduce implementation complexity and increase resilience to variation-related conservatism). We *customize* the leading core/thread specifically for lookahead. Our lookahead is not slowed down by unrelated normal computation, and the optimistic architectural design takes full advantage of correctness non-criticality. In contrast, the leading core/thread in these three different designs all execute almost the entire program and use a hardware that is designed with proper execution rather than lookahead in mind. Note that achieving clock speed improvement in the lead core as done in Tandem and Paceline is orthogonal to our design (which improves IPC) and can therefore be incorporated.

In Figure 12, we show the speedups. We note that our reproduction of DCE based on the paper is only a best-effort approximation. Not all details can be obtained, such as the baseline's prefetcher design. We used our own instead. In order not to inflate the benefit of EDA, we chose the best configuration for a sophisticated stream prefetcher [22]. Without this prefetcher, the baseline performance will be lowered by an average of 9% and as much as 25%. Finally, as mentioned before, improving simulator fidelity allows sometimes dramatic performance differences (*e.g.*, 3x in *swim*). Without this change, our version of DCE obtains performance improvements that match [9] relatively well. For better direct comparison, we

shift the basis of normalization in Figure 12 to the suboptimal baseline and show our default baseline as the "optimized" baseline. Our lookahead-specific design understandably provides more performance boosting.

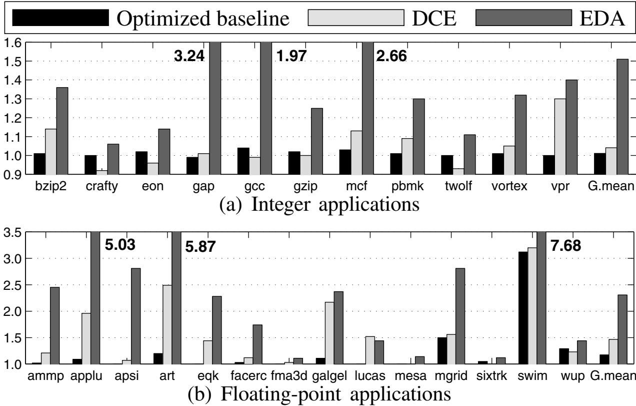


Figure 12. Comparison of EDA and DCE. All results are shown as speedup relative to the suboptimal simulator baseline. The optimized baseline, which has been used throughout the paper, is also shown.

Recap. In this section, we have shown that an explicitly-decoupled implementation can competently perform lookahead and deliver solid performance improvement. This is done without requiring complex circuitry that is challenging to implement. Explicit decoupling is a key enabling factor. Note that EDA does have an up-front cost in infrastructure. However, with further exploration, more novel techniques can be integrated to amortize the cost.

6. Related Work

By separating correctness guarantee and common-case performance optimization, our explicitly-decoupled architecture (EDA) has a number of key benefits: (a) improving the system's overall robustness by moving complexity of optimization out of the critical correctness core and (b) improving the efficiency of optimizations by allowing all layers in the design stack of the optimistic core to be removed from concerns of uncommon cases. With respect to the former, the pioneering work of DIVA is the most closely related [3]. A key difference is that our EDA argues for a far more decoupled implementation. The optimistic core communicates with the correctness core in an explicit fashion with low bandwidth, which allows the optimistic core to be in its own implementation domain with potentially different CAD strategies (*e.g.*, less conservatism in timing analysis) and voltage and frequency settings (less conservatism in PVT variation tolerance). Such benefits are much more difficult to obtain in tightly coupled systems such as DIVA or Razor [23].

Our EDA allows optimizations to be done in an optimistic and probabilistic fashion which we showed to be very effective. Of course, the goal of alleviating performance hurdles from memory access and branch misprediction is the same with a very wide set of approaches. But our way of achieving the gain has important differences with prior art.

In Section 5.3, we contrasted our approach with a class of designs using two passes to process a thread, including Slip-stream [7], [20], dual-core execution [9], Flea-flicker [21], Tandem [18], and Paceline [19]. Another class of related work is helper-threading (also called speculative precomputation) (*e.g.*, [24]–[32]). In helper-threading, a compiler- (or manually-) generated snippet of code is triggered at certain moments to access data in advance of the main thread. There are several challenges in cost-effective helper-threading. First, systematic and automatic generation of high-quality helper thread code is difficult. The code has to balance efficiency with success rate. Recall that these codes need to compute irregular addresses and they are invoked much higher up in the control flow when some necessary input may not be available. Duplicating the code to generate these data may be inefficient. Second, triggering helper threads at the opportune moment is also challenging. Both early and late triggering reduce a helper thread's effectiveness. Finally and perhaps most importantly, helper-threading needs the support to quickly and frequently communicate initial values at the register level from the main thread to the helper threads, which dictates that the hardware support for helper threads have to be tightly coupled to the core.

Our EDA-based approach avoids these challenges. First, our optimistic thread is automatically generated with a very simple parsing algorithm. Second, the optimistic thread is a continuous stand-alone thread. The correctness thread (software) does not spawn, control, or interfere with the optimistic thread. The correctness core (hardware) also has little additional complexity to enable or facilitate the optimistic thread. The only support involved is to pass on the register state at recovery time, which is done at an exceedingly low frequency. Moreover, the correctness thread is completely stalled during that transfer and thus there is no contention of any resource with normal processing. Finally, the distance between the pair of threads is trivially capped by the simple branch outcome queue, easily avoiding run-away lookahead. Moreover, delayed deployment of prefetching is straightforward in reducing premature prefetching.

To maintain high performance for the lead thread, we inevitably need to develop support to *tolerate* long latencies. A large body of work focuses on enhancing the processor's capability to buffer more in-flight instructions so as to avoid stalling [33]–[38], or to perform a special "runahead" execution during a conventional stall [12], [39]–[41]. In contrast, our lookahead is more proactive and continuous.

Finally, our work focuses on *explicitly* separating out performance-enhancing mechanisms which allows cost-effective implementations throughout the design stack (in a different domain). This paper is limited to ILP-enhancing assistive techniques. Extending the study to cover parallelization-oriented mechanisms such as [5] and [42] is our future work.

7. Conclusions and Future Work

In this paper, we have introduced performance-correctness *explicitly-decoupled architecture* with two separate domains,

each focuses only on one goal, performance optimization or correctness guarantee. By explicitly separating the two goals, both can be achieved more efficiently with less complexity.

Given the correctness guarantee, the performance domain can truly focus on the common case and allow the entire design stack to carry out optimizations in a very optimistic and efficient manner, avoiding excessive conservatism. With much complexity of optimization moved into the performance domain, the correctness domain can have a simpler architecture and circuit implementation and is thus easier to be made more robust against various emerging concerns such as PVT variations.

We have demonstrated a concrete design using two independent cores: the optimistic and the correctness core. We showed that both the microarchitecture of the optimistic core and its software can be designed optimistically and allow much simpler methods for optimizations. Such a design enables efficient, deep lookahead and produces a significant performance boosting effect (with geometric means of 1.49 and 1.96 speedup on integer and floating-point applications). It also allows the correctness core to be less aggressive in its implementation with less impacts on performance than such simplifications would have on a conventional microarchitecture.

While our first-step effort demonstrated some potentials of explicitly-decoupled architecture, there are a lot more to be explored. For instance, making the optimistic thread go beyond optimization of branch and cache misses; exploring more efficient, throughput-optimized design for the correctness core; conducting a more thorough design space study; and creating an intelligent and dynamic feedback system to allow a more targeted boosting effort. In future work, we plan to explore some of these areas.

References

[1] C. Bazeghi, F. Mesa-Martinez, and J. Renau, "μComplexity: Estimating Processor Design Effort," in *Proc. Int'l Symp. on Microarch.*, Dec. 2005.

[2] S. Borkar *et al.*, "Parameter Variations and Impact on Circuits and Microarchitecture," in *Proc. Design Automation Conf.*, Jun. 2003, pp. 338–342.

[3] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proc. Int'l Symp. on Microarch.*, Nov. 1999, pp. 196–207.

[4] M. Martin, M. Hill, and D. Wood, "Token Coherence: Decoupling Performance and Correctness," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2003, pp. 182–193.

[5] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," in *Proc. Int'l Symp. on Microarch.*, Nov. 2002, pp. 85–96.

[6] J. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 289–308, Nov. 1984.

[7] 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*, Nov. 2000, pp. 257–268.

[8] A. Garg and M. Huang, "A Performance-Correctness Explicitly-Decoupled Architecture," Department of Electrical & Computer Engineering, University of Rochester, Technical Report, Sep. 2008.

[9] H. Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window," in *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*, Sep. 2005, pp. 231–242.

[10] R. Muth *et al.*, "`alto`: A Link-Time Optimizer for the Compaq Alpha," *Software: Practices and Experience*, vol. 31, no. 1, pp. 67–101, Jan. 2001.

[11] O. Mutlu, K. Hyesoon, and Y. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," in *Proc. Int'l Symp. on Microarch.*, Dec. 2005, pp. 233–244.

[12] O. Mutlu *et al.*, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, Feb. 2003, pp. 129–140.

[13] *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Compaq Computer Corporation, Sep. 2000, order number: DS-0027B-TE.

[14] I. Kim and M. Lipasti, "Understanding Scheduling Replay Schemes," in *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, Feb. 2004, pp. 198–209.

[15] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Computer Sciences Department, University of Wisconsin-Madison, Technical Report 1342, Jun. 1997.

[16] S. Sair and M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suite," IBM T. J. Watson Research Center, Technical Report, Oct. 2000.

[17] J. Tendler *et al.*, "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, Jan. 2002.

[18] F. Mesa-Martinez and J. Renau, "Effective Optimistic-Checker Tandem Core Design Through Architectural Pruning," in *Proc. Int'l Symp. on Microarch.*, Dec. 2007, pp. 236–248.

[19] 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*, Sep. 2007, pp. 213–224.

[20] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," in *Proc. Int'l Symp. on Microarch.*, Dec. 2000, pp. 269–280.

[21] R. Barnes *et al.*, "Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining," in *Proc. Int'l Symp. on Microarch.*, Dec. 2003, pp. 387–398.

[22] 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*, Jun. 2005, pp. 54–61.

[23] T. Austin *et al.*, "Making Typical Silicon Matter with Razor," *IEEE Computer*, vol. 37, no. 3, pp. 57–65, Mar. 2004.

[24] M. Dubois and Y. Song, "Assisted execution," Department of Electrical Engineering, University of Southern California, Technical Report, 1998.

[25] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2001, pp. 52–61.

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

[27] C. Zilles and G. Sohi, "Execution-Based Prediction Using Speculative Slices," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2001, pp. 2–13.

[28] R. Chappell *et al.*, "Simultaneous Subordinate Microthreading (SSMT)," in *Proc. Int'l Symp. on Comp. Arch.*, May 1999, pp. 186–195.

[29] J. Collins *et al.*, "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2001, pp. 14–25.

[30] A. Roth and G. Sohi, "Speculative Data-Driven Multithreading," in *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, Jan. 2001, pp. 37–48.

[31] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi, "Slice-processors: an Implementation of Operation-Based Prediction," in *Proc. Int'l Conf. on Supercomputing*, Jun. 2001, pp. 321–334.

[32] A. Farcy *et al.*, "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes," in *Proc. Int'l Symp. on Microarch.*, Nov.–Dec. 1998, pp. 59–68.

[33] R. Balasubramonian *et al.*, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," in *Proc. Int'l Symp. on Microarch.*, Dec. 2000, pp. 245–257.

[34] A. Lebeck *et al.*, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *Proc. Int'l Symp. on Comp. Arch.*, May 2002, pp. 59–70.

[35] E. Torres *et al.*, "Store Buffer Design in First-Level Multibanked Data Caches," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2005.

[36] A. Gandhi *et al.*, "Scalable Load and Store Processing in Latency Tolerant Processors," in *Proc. Int'l Symp. on Comp. Arch.*, Jun. 2005, pp. 446–457.

[37] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Proc. Int'l Symp. on Microarch.*, Dec. 2003, pp. 423–434.

[38] S. Sethumadhavan *et al.*, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *Proc. Int'l Symp. on Microarch.*, Dec. 2003, pp. 399–410.

[39] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," in *Proc. Int'l Conf. on Supercomputing*, Jul. 1997, pp. 68–75.

[40] L. Ceze *et al.*, "CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction," *IEEE TCCA Computer Architecture Letters*, vol. 3, Dec. 2004.

[41] N. Kirman *et al.*, "Checkpointed Early Load Retirement," in *Proc. Int'l Symp. on High-Perf. Comp. Arch.*, Feb. 2005, pp. 16–27.

[42] 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.*, Jun. 2006, pp. 302–313.