AMNESIAC: Amnesic Automatic Computer

Trading Computation for Communication for Energy Efficiency *

Ismail Akturk Ulya R. Karpuzcu
University of Minnesota, Twin Cities {aktur002,ukarpuzc}@umn.edu

Abstract

Due to imbalances in technology scaling, the energy consumption of data storage and communication by far exceeds the energy consumption of actual data production, i.e., computation. As a consequence, recomputing data can become more energy-efficient than storing and retrieving precomputed data. At the same time, recomputation can relax the pressure on the memory hierarchy and the communication bandwidth. This study hence assesses the energy efficiency prospects of trading computation for communication. We introduce an illustrative proof-of-concept design, identify practical limitations, and provide design guidelines.

1. Motivation

Technology scaling and innovative architecture-level solutions to date have improved the energy efficiency of data generation, i.e., computation, significantly more than the energy efficiency of data communication [3, 14]. As a result, both, time and power spent in communication highly exceed the time and power spent in computation. Table 1, adapted from [18], compares the energy (time $\times$ power) consumption of an 64-bit load from on-chip SRAM (as a proxy for communication) and of a double-precision fused-multiply-add, FMA (as a proxy for computation), over two technology generations: 40nm and 10nm optimized for high performance (HP), and low power (LP), respectively. The communication energy increases from 1.55$\times$ computation energy at 40nm to approximately 6$\times$ at 10nm. Worse, off-chip communication to main memory requires more than 50$\times$ computation energy even at 40nm [18]. Off-chip memory accesses become more critical as data sets of emerging application domains keep growing.

Technology Node	**40nm**	**10nm**	
Operating Voltage	0.9V	0.75V (HP)	0.65V (LP)
Energy of 64-bit SRAM load (normalized to 64-bit FMA)	**1.55**	**5.75**	**5.77**

Table 1: Communication vs. computation energy [18].

As a consequence, recomputing data can become more energy-efficient than storing and retrieving precomputed data. In this paper, we hence investigate the effectiveness of recomputing data values in minimizing, if not eliminating, the overhead of expensive off-chip memory accesses. The idea is replacing a load with a sequence of instructions to recompute the respective data value, only if it is more energy-efficient. We call the resulting execution model *amnesic*[1] to contrast recomputation with conventional, *classic* execution.

Whether recomputation of a data value v can improve the energy efficiency or not tightly depends on where in the memory hierarchy the corresponding load would be serviced under classic execution, i.e., where in the memory hierarchy v resides. This is because the location of v in the memory hierarchy dictates the energy consumption of the respective load, $E_{ld,v}$, which in turn sets the energy budget for recomputation. Recomputation of v itself incurs an energy cost, $E_{rc,v}$, due to the (re)execution of the sequence of instructions to generate v. We will refer to each instruction in such a sequence as a *recomputing* instruction. Therefore, unless $E_{ld,v}$ exceeds $E_{rc,v}$, amnesic execution cannot improve energy efficiency.

Under amnesic execution, the sequence of recomputing instructions to generate v form a backward slice, which we will refer to as *recomputation slice*, *RSlice*. The first instruction in the slice is the immediate producer of v, $P(v)$. To be able to (re)execute $P(v)$, each input operand of $P(v)$ should be readily available at the anticipated time of recomputation. This may not always be the case, and (re)execution of $P(v)$ may trigger the re(execution) of producers of $P(v)$'s input operands, recursively.

The recomputation slice to generate v, *RSlice(v)*, can grow by tracking producer-consumer dependencies for recomputing instructions, however, not indefinitely. First of

* This work was supported by NSF CAREER CCF-1553042.

ASPLOS '17, April 08-12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037741

[1] amnesia [am'nēZHə]: noun, a partial or total loss of memory.
amnesiac [am'nēzē,ak], amnesic [-zik, -sik]: noun & adjective.

all, the energy cost of recomputation of v, $E_{rc,v}$, increases with the number of recomputing instructions in *RSlice(v)*, and amnesic execution cannot be energy-efficient if $E_{rc,v}$ exceeds the energy consumption of the respective load, $E_{ld,v}$. At the same time, not all of the input operands of recomputing instructions can be (re)generated by recomputation. This may be the case if input operands correspond to (i) read-only values to be loaded from memory, such as program inputs; or (ii) register values which are lost, i.e., overwritten at the time of recomputation.

Swapping loads for recomputation slices can reduce the pressure on memory bandwidth and unlock further opportunities for energy savings: For each load replaced with an *RSlice*, the corresponding store (to the same memory address) can become redundant if no other load (from the same address) depends on it. Therefore, amnesic execution can also filter out energy-hungry stores, and reduce the pressure on memory capacity by shrinking the memory footprint.

Under amnesic execution, the workload becomes more compute-intensive to make a better use of classic processors optimized for computation, as opposed to communication. In the following, we quantitatively characterize the energy efficiency potential of amnesic execution. This paper makes the following contributions:

- Introduction of a practical, illustrative proof-of-concept design to orchestrate *amnesic* execution.
- Identification and discussion of practical limitations and system-level implications of *amnesic* execution.

In the rest of the paper, Section 2 covers the semantics of amnesic execution; Section 3 details an illustrative proof-of-concept design; Sections 4 and 5 provide the evaluation; Section 6 compares and contrasts amnesic execution with related work, and Section 7 summarizes our findings.

2. Amnesic Execution Semantics

Under amnesic execution, an energy-hungry load is swapped with a sequence of recomputing instructions, which form a recomputation slice, *RSlice*, iff the energy cost of recomputation along the *RSlice* remains below the energy consumption of the respective load. In other words, the energy consumption of the load sets the energy budget for recomputation along the *RSlice*. If the anticipated energy cost of recomputation exceeds this budget, the respective load is performed and amnesic execution becomes equivalent to classic execution.

2.1 Recomputation Slice (*RSlice*)

For each data value v to be recomputed under amnesic execution, data dependences determine the order of the recomputing instructions in *RSlice(v)*. *RSlice(v)* includes the immediate producer instruction of v, $P(v)$, and possibly, producer instructions of the input operands of $P(v)$, in a recursive manner. Producer instructions may come from different basic blocks or functions.

Recomputation slices are very unlikely to comprise all producer instructions (i.e., producers of the producers) along a dependency chain, as the energy cost of recomputation along an *RSlice* increases with the number of recomputing instructions, and can easily exceed the energy consumption of the respective load. Amnesic execution prohibits recomputation in this case.

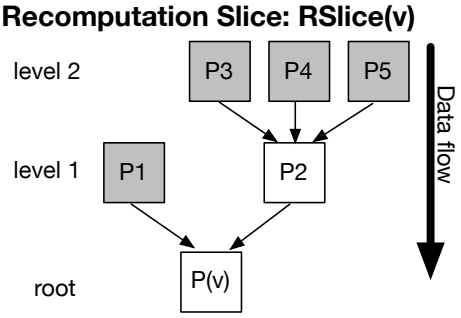

Figure 1: Example Recomputation Slice, *RSlice(v)*

Each recomputation slice, *RSlice(v)*, can be regarded as an upside-down tree with $P(v)$ residing at the root. Each node represents a producer instruction to be (re)executed. During recomputation along *RSlice(v)*, data flows from the leaves to the root. Figure 1 demonstrates an example. Nodes at level 1 correspond to immediate producers of the (input operands of the) root, nodes at level *l* correspond to the producers of nodes at level *l-1*. The number of incoming branches at each node reflects the number of producers of the node. Hence, *RSlice(v)* is not necessarily a balanced tree. As (re)executing only a finite number of nodes can fit into the energy budget set by $E_{ld,v}$, *RSlice(v)* cannot grow indefinitely. At the same time, the energy cost of recomputation along *RSlice(v)* includes the cost of retrieving input operands of the leaf nodes (which cannot rely on producers to recompute their inputs).

In the example from Figure 1, P1 and P2 at level 1 correspond to producers of $P(v)$'s input operands. (Re)execution of P1 does not require any more (re)execution. (Re)execution of P2, on the other hand, requires the (re)execution of three of P2's producers: P3, P4, and P5, respectively. The leaf producers are all shaded in gray. The leaves either represent terminal instructions which do not have any producers (e.g., instructions with constants as input operands), or instructions for which (re)execution of their producers is not energy-efficient. Amnesic execution can only function, if the input operands of leaf instructions are available at their anticipated time of (re)execution.

2.2 Non-recomputable Inputs

Not all of the input operands of leaf instructions of an *RSlice* can be (re)generated by recomputation. This may be the case if input operands correspond to (i) read-only values to be loaded from memory, such as program inputs; or (ii) register values which are lost, i.e., overwritten at the time of recomputation. We will refer to such input operands as *non-recomputable* inputs. For amnesic execution to work, non-recomputable inputs of *RSlice* leaves should not only be

available at the anticipated time of recomputation, but also be retrievable in an energy-efficient manner. Recomputation cannot eliminate any memory access to retrieve the non-recomputable inputs of *RSlice* leaves. If non-recomputable inputs do not reside in close physical proximity to the processor, the energy cost of their retrieval may easily exceed $E_{ld,v}$, rendering recomputation useless. In Section 3.2, we will discuss dedicated buffering for non-recomputable inputs. No dedicated buffering is necessary if the leaf input operands correspond to constants or live register values.

2.3 Side Effects

In this paper we focus on single-threaded amnesic execution[2]. Therefore, within the course of execution, recomputation along only one *RSlice* can be performed at a time. Amnesic execution should prevent corruption of the architectural state during recomputation, which can be achieved by allocating dedicated buffers (Section 3.2) similar to classic microarchitectural storage for speculative state.

Amnesic execution can orchestrate exception handling similar to exception handling under speculation, as well: record exceptions as long as recomputation along an *RSlice* is taking place, and defer their handling after recomputation finishes. However, we may need to revisit the definition of (im)precise exceptions in this case, since recomputation modifies the architectural control flow by executing extra (recomputing) instructions, as opposed to speculation.

3. An Illustrative Proof-Of-Concept Amnesic Implementation

The critical question under amnesic execution is *when to fire recomputation*. Potentially, the compiler can extract *RSlice(v)* for each load (to read v), by tracking data dependences. Whether recomputation along *RSlice(v)* is more energy-efficient than performing the respective load, however, depends on where in the memory hierarchy v resides. Being able to only speculate where v can reside during execution, the compiler can at most probabilistically estimate the energy consumption of the respective load, $E_{ld,v}$, which sets the energy budget for recomputation. For each v where recomputation is estimated to be more energy-efficient, the compiler can modify the binary to swap the load for *RSlice(v)*. In the following, we will discuss various implementation options and how microarchitectural support can help.

The basic proof-of-concept implementation covered in this section features an amnesic compiler (Section 3.1), microarchitectural support for amnesic execution (Section 3.2), and a runtime (instruction) scheduler to orchestrate amnesic execution (Section 3.3). We first let the compiler identify and annotate a set of independent recomputation slices. Then, at runtime, the amnesic scheduler fires or skips recomputation along each *RSlice(v)*, by tracking where in the memory hierarchy v resides at the anticipated time of recomputation.

3.1 Amnesic Compiler and Instruction Set Extensions

The amnesic compiler first extracts a set of independent *RSlice*s as potential targets for recomputation, and annotates each, such that the amnesic scheduler (Section 3.3) can identify them at runtime. The amnesic scheduler triggers recomputation along any given *RSlice(v)* only if loading the data value v is more energy-hungry than recomputation.

3.1.1 Slice Formation

The amnesic compiler pass first estimates, probabilistically (as detailed in the following and Section 4), the energy consumption of loading v, $E_{ld,v}$. Next comes dependency analysis to identify the producer instructions of v, in order to calculate the anticipated cost of potential recomputation. This step starts building *RSlice(v)* (where the immediate producer of v, $P(v)$, resides at the root), and lets *RSlice(v)* grow level by level, as long as the cumulative cost of recomputation along *RSlice(v)* being constructed remains below $E_{ld,v}$.

As the compiler traverses the dependency chains in constructing *RSlice(v)*, it may hit load instructions. In the proof-of-concept implementation, the compiler replaces each such load with the respective recomputing slice, recursively. Therefore, loads and stores cannot be present as intermediate nodes in *RSlice(v)*.

To derive the energy cost of recomputation, $E_{rc,v}$, the compiler pass uses instruction mix and count within *RSlice(v)*, along with machine specific energy per instruction (EPI) estimates: $E_{rc,v}$ is the sum of [*instruction count per category*] $\times$ [*EPI per category*], over all instruction categories represented in *RSlice(v)*'s instruction mix. $E_{ld,v}$ calculation, on the other hand, relies on probabilistic estimates: Pr_{Li}, the probability of having a load serviced by level *Li* in the memory hierarchy, is derived from hit and miss statistics of *Li* under profiling. Let the EPI estimate for a load serviced in *Li* be EPI_{Li}. Then, the sum of $\text{Pr}_{Li} \times \text{EPI}_{Li}$ over all levels i in the memory hierarchy (including off-chip) gives the probabilistic energy cost per load.

3.1.2 Slice Annotation

As a hint for the amnesic scheduler, the compiler replaces each load, the swap of which with recomputation is likely to be more energy-efficient (according to the probabilistic energy cost comparison explained above) with a special control flow instruction, `RCMP`. In this case, the compiler also inserts the constructed *RSlice* in the binary.

Semantically, `RCMP` corresponds to the fusion of a conditional branch with a load[3]. The resolution of the branching condition is left to the amnesic scheduler (Section 3.3) at runtime. Depending on the branching condition (which

[2] Under parallel execution, communication with memory expands along two dimensions: accesses to thread-local data and accesses to shared data. In this paper, we focus on the first, in the context of single-threaded execution. In principle, loads swapped for recomputation may be triggered by core-to/from-memory (thread-local) or core-to-core (shared) communication.

[3] Depending on the specifics of the underlying instruction set architecture (ISA), `RCMP` can also be synthesized by a pair of branch and load instructions, without loss of generality.

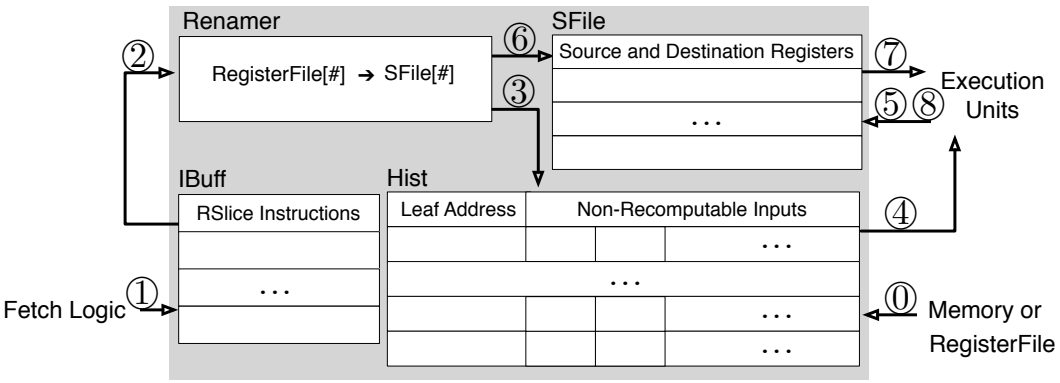

Figure 2: Amnesic Microarchitecture & Scheduler

is dictated by where in the memory hierarchy v resides at runtime), RCMP can act either as a branch to the entry point (starting from the leaves) of *RSlice(v)*, or as a classic load which reads v from memory. The latter is the case if the amnesic scheduler determines at runtime that recomputation is less energy-efficient than performing the load, i.e., $E_{rc,v}$ exceeds $E_{ld,v}$. Accordingly, as input operands, RCMP inherits all input operands of the respective load, in addition to the starting address of *RSlice(v)*.

At the exit of each such *RSlice(v)* embedded in the binary resides a return instruction, RTN, which returns the control to the instruction following RCMP in program order after recomputation along *RSlice(v)* finishes. RTN semantics closely mimic procedure return instructions. Before return, the recomputed data value v gets copied into the destination register of the eliminated load (recall that RCMP inherits all source and destination parameters of the respective load).

Only if the leaves of RSlice(v) have non-recomputable input operands, the compiler places REC instructions into the binary, which serve buffering of non-recomputable input operands such as overwritten register values. An REC instruction goes right after each instruction, a replica of which serves as a leaf in *RSlice(v)*. REC has a single integer operand: `leaf-address` which points to the address of the respective leaf instruction in *RSlice(v)*. REC practically checkpoints the input operands to a dedicated buffer (Sections 3.2 and 3.3).

Unless the compiler can prove that all input operands of *RSlice(v)*'s leaves correspond to constants or live register values at the anticipated time of recomputation, REC instructions are necessary. Finally, how the compiler orders the leaves in *RSlice(v)* code is not critical, as leaf instructions cannot depend on each other.

3.2 Amnesic Microarchitecture

Amnesic execution should meet two conditions for safe and effective recomputation:

Condition-I: Prevent corruption of the architectural state during recomputation (Section 2.3).

Condition-II: Have (non-recomputable) input operand values of *RSlice* leaves available at the anticipated time of recomputation (Section 2.2).

Fig. 2 captures microarchitectural support to meet **Condition-I** and **Condition-II** in orchestrating amnesic execution. Recall that only one *RSlice* can be active, i.e., traversed for recomputation, at a time[4].

Scratch-File (SFile): To satisfy **Condition-I**, the amnesic microarchitecture deploys the dedicated buffer SFile. During recomputation, as program control traverses an *RSlice*, the data flows through the SFile, leaving the (physical) registerfile intact. Recomputing instructions from an *RSlice* do not perform any memory access, and communicate over SFile only.

Renamer: During traversal of each *RSlice*, a dedicated Renamer maps register references per recomputing instruction to SFile entries. Semantically, the amnesic renamer closely mimics the rename logic of classic out-of-order machines. In this context, SFile becomes not any different than the physical registerfile and follows similar rules for space (de)allocation.

History Table (Hist): For each *RSlice* where the leaf input operands correspond to constants or live values from the (physical) registerfile, **Condition-II** is automatically satisfied. Only for non-recomputable leaf input operands, dedicated storage is required to satisfy **Condition-II**. The amnesic microarchitecture can buffer non-recomputable input operands for each *RSlice* leaf in the dedicated history table Hist. Each entry of Hist keeps the address (`leaf-address`) and non-recomputable input operands of a leaf instruction.

Instruction Buffer (IBuff) can cache recomputing instructions within each *RSlice*, in order to relax amnesic execution's potential pressure on the instruction cache. Each entry of IBuff corresponds to a recomputing instruction.

SFile, Hist, and IBuff all feature an `invalid` field per entry to orchestrate (de)allocation of space as necessary.

3.3 Amnesic Scheduler

3.3.1 Runtime Policies

At runtime, the amnesic scheduler decides whether recomputation along each *RSlice(v)* embedded into the binary by

[4] Offloading recomputation to spare or idle cores, or using helper threads may improve energy efficiency further by enabling concurrent recomputation. However, the basic proof-of-concept implementation assumes strictly sequential execution semantics.

the compiler (Section 3.1) can improve energy efficiency or not, depending on where in the memory hierarchy v resides. Specifically, each time a RCMP instruction is fetched, the scheduler has to decide whether to branch to the entry point of the respective *RSlice(v)*, or whether to perform the load to read v from memory. A control flag, `recompute`, remains set as recomputation – traversal of an *RSlice* – is in progress. `recompute` is reset by default.

To be able to draw a safe decision, the amnesic scheduler needs to track where in the memory hierarchy v resides. There are different options to track or predict the location of v at runtime. In the proof-of-concept implementation, the amnesic scheduler lets the corresponding load probe on-chip memory (caches), and fires recomputation upon a miss in the first-level cache (*FLC*), or alternatively, upon a miss in the last-level cache (*LLC*) – by using either a first or a last level cache miss as an indicator for an energy-hungry off-chip memory access. In this case, RCMP becomes the equivalent to `branch on FLC miss` or, alternatively, `branch on LLC miss`, with the branch target being the entry point of the respective *RSlice*. The amnesic scheduler fires recomputation by setting the `recompute` flag. Otherwise, execution follows the classic trajectory by performing the load.

In this case, recomputation cost includes the cost of probing the on-chip memory hierarchy. *FLC* and *LLC* policies are heuristic-based and may result in false-negatives (lost recomputation opportunity) and false-positives (energy-inefficient recomputation). Better amnesic policies can be devised by using more accurate (miss) predictors [28, 15, 1], which can also help eliminate the probing overhead. We leave further refinement and exploration of such policies to future work – the design space is pretty rich. In Section 5, we will also compare *FLC* and *LLC* policies to a runtime-oblivious policy, *Compiler*, which *always* triggers recomputation each time a RCMP instruction is fetched.

3.3.2 Putting It All Together

Amnesic activity when `recompute` is reset: No recomputation takes place as long as the `recompute` flag stays reset. During this period, amnesic execution is equivalent to classic execution, if no *RSlice* in the binary features non-recomputable leaf inputs. Otherwise, the amnesic scheduler has to record such non-recomputable input operands into Hist. To this end, the scheduler tracks REC instructions (Section 3.1.2). REC instructs the scheduler to record all non-recomputable input operands in a Hist entry (⓪ in Fig. 2), along with `leaf-address`.

Triggering recomputation: For each RCMP instruction fetched, the amnesic scheduler first needs to resolve the branching condition: whether recomputation is more energy-efficient than performing the memory access, i.e., whether $E_{ld,v}$ exceeds $E_{rc,v}$. This decision can be drawn following any of the runtime policies from Section 3.3.1, *FLC* or *LLC*. For example, under *LLC*, the amnesic scheduler probes the caches, and fires recomputation by setting the `recompute` flag upon an LLC miss. Otherwise, the load is performed following the classic execution trajectory.

Amnesic activity when `recompute` is set: RCMP branches to the entry point of *RSlice(v)*, and instruction fetch starts from the first leaf. Each leaf instruction first has its destination register renamed (② in Fig. 2). Each leaf instruction with non-recomputable input operands next probes Hist with `leaf-address` (③) to read its input operands, which directly are fed into the corresponding execution units (④). Leaf instructions with constant or live register input operands do not need to probe Hist. Upon finishing execution, each leaf writes its result to the SFile (⑤).

Non-leaf recomputing instructions which represent intermediate nodes in *RSlice(v)* read their input operands from SFile (⑥) after having their source and destination registers renamed (②). Upon collecting the input operands, recomputing instructions proceed to the execution units (⑦), and write their results back to the SFile once execution completes (⑧). All (non-leaf) recomputing instructions in *RSlice(v)* execute sequentially in this manner until the RTN instruction of the slice is fetched. Before return, the recomputed data value v gets copied from SFile into the destination register of the eliminated load (recall that RCMP inherits all source and destination parameters of the respective load). The amnesic scheduler then resets `recompute` flag to demarcate the end of recomputation. Execution continues from the instruction following RCMP in program order.

IBuff is an optional structure to help reduce the pressure on instruction cache under recomputation. Very much like the instruction cache, fetch logic can fill IBuff with recomputing instructions (①). IBuff in turn feeds the Renamer with recomputing instructions (②).

3.4 Storage Complexity

We next analyze the expected storage complexity for each component of the amnesic microarchitecture from Fig. 2. Recall that the amnesic microarchitecture only processes instructions with register source operands and register destinations, and excludes memory or control flow instructions. Without loss of generality, the following analysis assumes a RISC-style ISA.

SFile: A recomputing instruction typically writes its result to one destination register, and reads its input operands from two source registers. Accordingly, the maximum possible number of renaming requests per recomputing instruction, $max_{\#\text{rename}}$ becomes

$$max_{\#\text{rename}} = max_{\#\text{src}} + max_{\#\text{dest}} = 3$$

where $max_{\#\text{src}}$ ($max_{\#\text{dest}}$) is the maximum number of source (destination) register operands per recomputing instruction. At any given time, only one *RSlice* can be traversed. Therefore, SFile capacity does not depend on the total *RSlice* count in the binary, but grows with the instruction count per *RSlice*, which can exponentially increase with the tree height h. A tall *RSlice*, however, is very unlikely to find any place in the binary, as it can easily result in excessive recomputation overhead to render recomputation

useless. The amnesic compiler captures such diminishing returns and prevents excessive growth of the *RSlice* (Section 3.1): practically, the compiler not only influences *RSlice* topology, but also caps the tree height h to maximize energy savings. Accordingly, we can derive a loose upper-bound for SFile capacity as

$$max_{\#\text{inst per } RSlice} \times max_{\#\text{rename}} = max_{\#\text{inst per } RSlice} \times 3$$

where $max_{\#\text{inst per } RSlice}$ corresponds to the maximum of instruction count per *RSlice* across all *RSlice*s in the binary.

Hist: Hist can keep data for multiple *RSlice*s during execution. For each *RSlice*, Hist can contain as many entries as the *RSlice*'s number of leaves. Thus, a loose upper-bound for the number of entries in Hist becomes

$$\#RSlice \times max_{\#\text{leaf per } RSlice}$$

where $\#RSlice$ is the number of *RSlice*s in the binary; and $max_{\#\text{leaf per } RSlice}$, the maximum of the number of leaves per *RSlice* (which may grow with tree height h). Each Hist entry accommodates at most $max_{\#\text{src}}$ values, to cover all non-recomputable input operands per leaf.

IBuff: The capacity of IBuff grows with the number of instructions per *RSlice*. Hence, a loose upper-bound for IBuff capacity becomes $max_{\#\text{inst per } RSlice}$.

3.5 Technicalities

The proof-of-concept implementation represents a basic design, which neglects various optimization opportunities such as instruction reuse among recomputing slices, or hardware resource sharing with the underlying microarchitecture.

During traversal of an *RSlice*, latency per recomputing instruction remains very similar to its classic counterpart, as the amnesic microarchitecture follows the pipelining semantics of the underlying microarchitecture (just with an alternative instruction and operand supply of similar latency).

The storage complexity of amnesic structures from Fig. 2 tends to be low (Section 3.4). Only the unlikely capacity overflow of Hist can impair recomputation, and only for *RSlice*s with non-recomputable leaf input operands. The amnesic scheduler can track these cases by failed `REC` instructions (Section 3.1.2) and enforce the corresponding `RCMP` to skip recomputation (i.e., to perform the load). To this end, the amnesic scheduler has to uniquely identify the matching `RCMP`. This can be achieved by assigning a unique ID, `RSlice-ID`, to each *RSlice* in the binary, and providing it as an operand to both `REC` and `RCMP`.

In processing recomputing instructions, the amnesic microarchitecture has to differentiate between leaves and intermediate nodes, since different structures supply the input source operands to each: The inputs of leaves can come from the registerfile (a live value) or Hist (an overwritten value). The inputs of intermediate nodes come from SFile. The compiler annotates leaves and accesses to Hist to distinguish between these cases. Specifically, the compiler changes source register identifiers of leaf instructions reading their operands from Hist to an invalid number. Leaf instructions with valid source register identifiers directly access the registerfile. Non-leaf recomputing instructions follow the paths ② and ⑥ in Fig. 2.

Recall that there is another potential class of leaves with non-recomputable input operands: read-only values to be loaded from memory, such as program inputs. In principle, replacing the load to read v from memory with *RSlice(v)* which features possibly more than one such load at the leaves does not make sense. Hist is designated to record overwritten register input operands, but Hist can also keep such read-only values, and may make recomputation along such *RSlice(v)* energy-efficient.

4. Evaluation Setup

Benchmarks: To quantify the energy efficiency potential of amnesic execution, we experiment with 33 sequential or single-threaded benchmarks from SPEC-2006 [13], NAS [2], PARSEC [4] and Rodinia [8] suites, which span various application domains and memory access characteristics, as listed in Table 2.

Suite	Benchmarks	Inputs
SPEC	mcf, perlbench, gobmk, calculix, GemsFDTD, libquantum, soplex, lbm, omnetpp, sphinx3 (sx)	test
NAS	is	A
	cg	W
	ft, mg	S
PARSEC	canneal (ca), facesim (fs), ferret (fe), raytrace (rt), blackscholes, x264, dedup, freqmine, fluidanimate, streamcluster, swaptions, bodytrack	simsmall
Rodinia	backpropagation (bp)	65536
	bfs	graph1MW_6.txt
	kmeans	kdd_cup
	nw	2048 10 1
	particlefilter	-x 128 -y 128 -z 10 -np 10000
	srad (sr)	100 0.5 502 458 1
	hotspot	512 512 2 1

Table 2: Benchmarks deployed.

Binary generation: We implement the greedy compiler pass detailed in Section 3.1 as a (*binary generator*) Pin [25] tool. The EPI estimates (Section 3.1.1) come from measured data from [33]. Although these estimates are for a parallel processor (Intel's Xeon Phi), the simulated microarchitecture is very similar to its per core configuration (Table 3). We also fine-tune these estimates by extracting EPI values for different instruction categories from McPAT [23] integrated with the Sniper-6.1 [7] microarchitectural simulator. We derive Pr_{Li} (Section 3.1.1), the probability of having a load serviced by level *Li* in the memory hierarchy, using hit and miss statistics for *Li* from Sniper. We also implement a *runtime profiler* in Pin, which collects dependency information for binary generation. Using the dependency information (from

the Pin-based runtime profiler) and EPI estimates, the (binary generator) Pin tool identifies *RSlice*s that can improve energy efficiency, and instruments them for inclusion into the binary.

Technology node:		22nm	
Operating frequency:		1.09 GHz	
L1-I (LRU):	32KB, 4-way	0.88nJ	3.66ns
L1-D (LRU, WB):	32KB, 8-way	0.88nJ	3.66ns
L2 (LRU, WB):	512KB, 8-way	7.72nJ	24.77ns
Main Memory	Read: 52.14nJ	Write: 62.14nJ	100ns

Table 3: Simulated architecture.

Recomputation at runtime: We implement the amnesic microarchitecture from Fig. 2 in Sniper, and run the annotated binaries on it. Sniper facilitates seamless integration with Pin. Runtime energy and performance statistics come from Sniper (+ McPAT) simulations. Table 3 gives EPI and (round-trip) access latency for each level in the simulated memory hierarchy. We conservatively model EPI and access latency for Hist after L1-D; for SFile, after the physical registerfile; and for IBuff, after L1-I. Accordingly, we model `RCMP`'s overhead after a conditional branch; `REC`'s, after a store to L1-D; `RET`'s, after a jump.

5. Evaluation

5.1 Impact on Energy Efficiency

Fig. 3 captures the impact of amnesic execution on energy-delay product, EDP [11], as a proxy for energy efficiency. The y-axis is normalized to the EDP under classic execution. Out of 33 benchmarks we deployed, only 11 have the potential to provide more than 10% EDP gain. In the following, we will focus on these benchmarks. The rest of the benchmarks did not benefit much from recomputation (only 4 provided more than 5% EDP gain) because they did not have many energy-hungry loads and/or recomputation degraded temporal locality. Recomputation cannot improve energy efficiency of compute-bound applications unless they incorporate a few but very energy-hungry memory references.

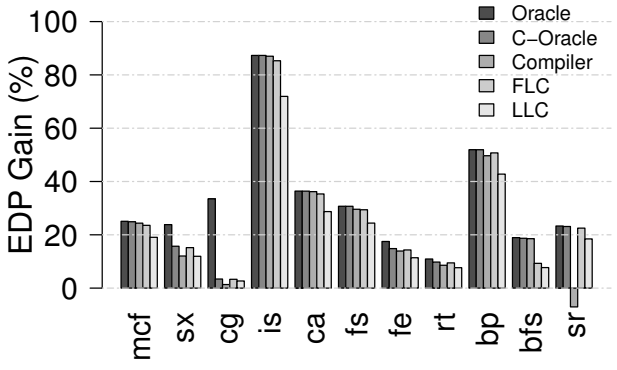

Figure 3: EDP gain under amnesic execution.

In Fig. 3, we compare representative runtime policies from Section 3.3.1 – *FLC*, *LLC* and *Compiler*, to two oracular policies: *Oracle* and *C(onservative)-Oracle*. *FLC*, *LLC*, *Compiler* and *C-Oracle* select from the very same set of *RSlice*s for recomputation at runtime – this set is identified by the compiler pass using the probabilistic energy model (Section 3.1.1). At runtime, *FLC* (*LLC*) fire recomputation along *RSlice(v)* if the respective load of v misses in FLC (LLC). *Compiler*, on the other hand, *always* fires recomputation, for each `RCMP` encountered.

C-Oracle can predict with 100% accuracy where the load of v will be serviced in the memory hierarchy as the amnesic scheduler decides whether to perform the load or whether to fire recomputation along *RSlice(v)*. *C-Oracle* hence bases the runtime decision on this 100% accurate prediction. *Oracle*, too, can predict at runtime with 100% accuracy where a load would be serviced. The key difference of *Oracle* from *C-Oracle* comes from a different (i.e., optimal) set of *RSlice*s baked in the binary, than the compiler's probabilistic energy model based set (which applies to the rest of the policies). The EDP difference between *Oracle* and *C-Oracle* therefore illustrates how accurate compiler's probabilistic energy model is. The smaller the EDP difference, the more accurate is the probabilistic energy model in characterizing an application's loads. In other words, *C-Oracle* demonstrates the maximum possible EDP gain with the given probabilistic energy model of the loads.

We fine-tune the probabilistic energy model of the amnesic compiler pass using dynamic execution traces (Section 3.1.1). Notice that the EDP gain under *Compiler* evolves with the accuracy of this probabilistic energy model, but such fine-tuning may not always be possible. The more accurate the energy model, the more accurate becomes amnesic compiler's prediction of where the load reading v will be serviced at runtime. And the more accurate this prediction, the more energy efficiency can the *Compiler* policy harvest, under which each `RCMP` always triggers recomputation. The EDP gains under *Compiler* therefore reflect best-case estimates.

Recall that the set of *RSlice*s recomputed by each policy is different: *Compiler* recomputes along each *RSlice* embedded in the binary, which form the set S. *C-Oracle* picks the optimal subset from S ($S_{C-Oracle}$) for recomputation, i.e., only recomputes *RSlice(v)* if recomputation is exactly more energy-efficient than performing the load of v. *FLC* (*LLC*), on the other hand, picks the subset of S, S_{FLC} (S_{LLC}), which only includes *RSlice(v)*s where the respective load to read v misses in L1 (L2). Subject to the accuracy of the probabilistic energy model and such runtime decisions, the set of *RSlice*s recomputed by *Oracle* may be very different: *Oracle*'s decisions are based on actual (not probabilistic or predicted) energy costs.

Overall, with the exception of sx and cg (and fe, rt to a lower extent), we observe that *C-Oracle* closely tracks *Oracle*, rendering the probabilistic energy model accurate. Except sr, the best-case *Compiler* closely tracks *C-Oracle*. On the other hand, the difference between the best-case *Compiler* and *FLC* is barely visible, with the exception of sx, bfs and sr. *LLC* is consistently worse than *FLC*. The main

delimiter for *LLC* is the overhead of probing the last-level cache (L2) to detect a miss which is much larger than the overhead of probing the first-level cache (L1) to detect a miss under *FLC*.

$\mathbf{EDP(Compiler)} < \mathbf{EDP(FLC)}$: In principle, as the amnesic compiler can only probabilistically take into account where a load might get serviced at runtime, by firing recomputation along *RSlice(v)* for each `RCMP` encountered, the *Compiler* policy can easily trigger unnecessary recomputations, and hence, hurt energy efficiency – particularly if v resides in L1. *FLC*, on the other hand, prevents recomputation in this case. This is clearly visible for sr, where *Compiler* triggers too many recomputations that do not provide sizable energy gain (due to recomputed data mostly being in L1), but introduce performance overhead (since RSlices recomputed usually take longer than accessing L1). Since the energy gain due to recomputation does not offset the performance degradation, the EDP of sr degrades 7% under *Compiler*. Although the difference is small, *Compiler* yields lower EDP gain than *FLC* in sx, cg, fe, rt and bp.

$\mathbf{EDP(Compiler)} > \mathbf{EDP(FLC)}$: *Compiler* can provide higher gains than *FLC* (*LLC*) when they recompute the very same set of *RSlice*s; i.e., S_{FLC} (S_{LLC}) overlaps with S – when none of the vs is present in L1 (L2). This is because *Compiler* does not need to probe the caches, so there is no probing cost. Although the difference is mostly small, this is the tendency in mcf, is, ca, fs, and bfs.

$\mathbf{EDP(FLC)}$ **vs.** $\mathbf{EDP(LLC)}$: If v resides in L1, both *FLC* and *LLC* simply skip recomputation. If v resides in L2, only *FLC* fires recomputation. In this case, depending on the instruction mix and count in *RSlice(v)*, recomputation may be less expensive than retrieving v from L2, particularly for short *RSlice(v)*. At the same time, the probing cost is lower for *FLC* than *LLC*. As Section 5.4 reveals, the benchmark applications feature predominantly short *RSlice(v)*s, with much less than 50 instructions. Overall, *FLC* renders the higher EDP gain, since recomputation along *RSlice(v)* remains usually cheaper than retrieving v from L2.

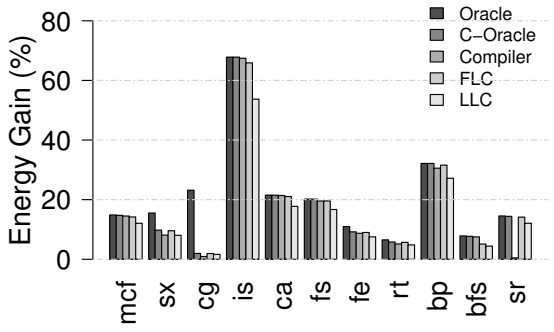

Figure 4: Energy gain under amnesic execution.

Impact on energy & execution time: Due to memory accesses being both energy-hungry and slow, most of the time, the reduction in EDP comes from a reduction in both energy and execution time. Fig. 4 shows the corresponding reduction in energy consumption; Fig. 5, in execution time, under amnesic execution, normalized to classic execution. We observe similar trends to EDP for both.

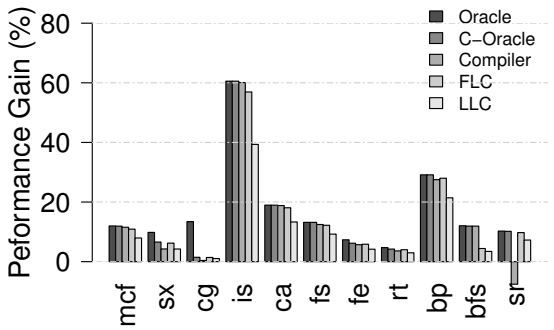

Figure 5: % reduction in execution time.

Putting it all together: An amnesic design which always fires recomputation following compiler hints (i.e., *Compiler*, as opposed to following policies like *FLC* or *LLC*) can be very effective as Fig. 3 reveals, but it is limited by the accuracy of compiler's probabilistic energy model. Overall, *Compiler* improves the EDP of all benchmarks, with the exception of sr and mg where EDP is degraded by 7% and 1.37%, respectively. Eight of the benchmarks obtain more than 10% EDP gain under *Compiler*, where the range changes from 12.04% to 87%. *FLC* and *LLC* yield slightly lower EDP gains than *Compiler*, in general. Since they tend do make more conservative decisions on recomputation, they do not experience any EDP degradation. For the afore-mentioned 8 benchmarks, EDP gain under *FLC* (*LLC*) range from 14.37% to 85.3% (11.39% to 71.92%).

To shed further light on these findings, we will next look into instruction mix (Section 5.2), memory access characteristics (Section 5.3) and *RSlice* characteristics (Section 5.4) under amnesic execution.

5.2 Impact on Instruction Count and Mix

Under amnesic execution, the sequence of recomputing instructions in each *RSlice(v)* replaces the respective load to read v from memory. Therefore, we expect an increase in the number of (dynamic) instructions along with a decrease in the number of (dynamic) load instructions under amnesic execution. Table 4 shows how the dynamic instruction mix and energy breakdown changes under amnesic execution. For comparison, we also provide the energy breakdown under classic execution. Without loss of generality, we report the amnesic execution outcome for the *Compiler* policy, which incurs the maximum possible number of recomputations.

The first half of the table captures the % increase in the dynamic instruction count along with the % decrease in the dynamic load count under amnesic execution with respect to the classic baseline. In the second half, we report the % energy breakdown under classic and amnesic execution: we differentiate between stores, loads and all other instructions (which form the category *Non-mem*). Under amnesic execution, we also report the share of Hist table reads, which retrieve non-recomputable input operands of *RSlice* leaves.

Benchmark	% increase in (dyn.) instruction count	% decrease in load count	Classic Energy Breakdown (%)			Amnesic Energy Breakdown (%)			
			Load	Store	Non-mem	Load	Store	Non-mem	Hist Read
mcf	4.47	6.19	91.67	2.12	6.20	75.33	2.88	6.77	0.48
sx	4.55	6.68	70.43	2.70	26.86	58.44	3	28.01	2.42
cg	3.97	2.11	82.43	0.45	17.10	80.03	0.51	17.99	0.51
is	17.97	49.99	84.30	11.19	4.49	9.62	13.17	9.75	3.06e-06
ca	7.38	7.95	85.21	5.16	9.61	62.26	5.20	10.42	0.70
fs	1.83	3.08	53.90	14.37	31.71	32.36	14.78	32.61	0.68
fe	3.55	1.75	58.49	15.50	26	47.81	15.57	27.03	0.84
rt	1.97	6.08	67.87	8.58	23.54	60.67	8.73	24.27	1.16
bp	31.89	55.55	87.71	7.22	5.05	52.68	7.22	7.38	2.13
bfs	1.20	60.93	79.18	1.87	18.94	68.35	2.20	21.92	2.42e-07
sr	20.02	23.33	49.89	9.43	40.66	30.35	14.69	47.11	7.36

Table 4: Dynamic instruction mix and energy breakdown under amnesic execution.

Benchmark	Compiler			FLC			LLC		
	L1-hit (%)	L2-hit (%)	Memory-hit (%)	L1-hit (%)	L2-hit (%)	Memory-hit (%)	L1-hit (%)	L2-hit (%)	Memory-hit (%)
mcf	12.02	11.01	76.97	10.73	11.16	78.09	10.73	11.16	78.09
sx	85.33	0.85	13.80	85.08	0.86	14.04	85.09	0.85	14.05
cg	87.49	0.17	12.33	87.49	0.17	12.33	87.49	0.17	12.33
is	49.64	19.25	31.10	49.64	19.25	31.10	49.64	19.25	31.10
ca	27.85	7.50	64.63	27.84	7.51	64.64	27.84	7.51	64.64
fs	56.47	1.92	41.59	56.46	1.92	41.60	56.46	1.92	41.61
fe	63.26	10.06	26.67	63.22	10.07	26.70	63.22	10.05	26.71
rt	92.95	0.75	6.28	92.21	0.83	6.94	92.85	0.06	7.07
bp	72.49	4.11e-3	27.49	72.49	4.11e-3	27.49	72.49	4.11e-3	27.49
bfs	98.43	1.15e-3	1.56	98.43	1.15e-3	1.56	98.43	1.15e-3	1.56
sr	93.70	0.03	6.26	93.70	0.03	6.26	93.70	0.03	6.26

Table 5: Memory access profile of load instructions under classic execution, which are swapped for recomputation under *Compiler*, *FLC*, and *LLC* policies, respectively.

We observe that amnesic execution reduces the energy consumed by load instructions for all benchmarks, while the energy consumed by *Non-mem* instructions increases due to recomputation along *RSlice*s. is from NAS, among the benchmarks listed in Table 4, is the most responsive to amnesic execution: The energy consumption of its loads drops from 84.3% to 9.62%, at the expense of executing $\approx$ 17.97% more instructions due to recomputation. In return, the number of dynamic loads reduces by 49.99% under amnesic execution.

5.3 Memory Access Characteristics

The effectiveness of amnesic execution is constrained by, for each target data value v, (i) where in the memory hierarchy v resides; (ii) the cost of recomputation along *RSlice(v)*. (i) sets the budget for recomputation, and recomputation is only effective if (ii) remains below this budget. The lower the level in the memory hierarchy where v resides, the higher becomes the budget for recomputation along *RSlice(v)*. Amnesic execution is more likely to provide higher energy efficiency, if the target v resides in lower levels of the memory hierarchy.

Table 5 shows the memory access profile of load instructions under classic execution, which are swapped for recomputation under *Compiler*, *FLC*, and *LLC* policies, respectively. We report the percentage of such load instructions serviced by each level in the simulated memory hierarchy (Table 3). Recall that the set of *RSlice*s recomputed by each policy is different (Section 5.1), therefore, so is the set of loads swapped for recomputation.

Memory access characteristics help us reason about why some benchmarks benefit more from recomputation, considering different policies. For example, bfs exhibits higher EDP gain for the *Compiler* policy, but relatively lower EDP gain for *FLC* and *LLC* policies (Fig. 3). As Table 5 reveals, bfs's swapped loads are almost entirely serviced by L1. Since bfs's swapped loads barely miss in L1, *FLC* and *LLC* policies fire recomputation less often. *Compiler*, on the other hand, triggers recomputation regardless of where the target data resides in the memory hierarchy. bfs's energy efficiency gain under *Compiler* comes from the relatively short, hence cheap *RSlice(v)*s (Section 5.4), even though the target v could be found in L1 most of the time. In this case, *Compiler* comes very to close *Oracle*.

Quite the opposite trend applies for sr, the benchmark where *Compiler* falls noticeably behind *Oracle* and even degrades the EDP. As Table 5 reveals, similar to bfs, most (93.7%) of sr's swapped loads are serviced by L1. As it was the case for bfs, the target v could be found in L1 most of the time, but *Compiler* always triggers recomputation

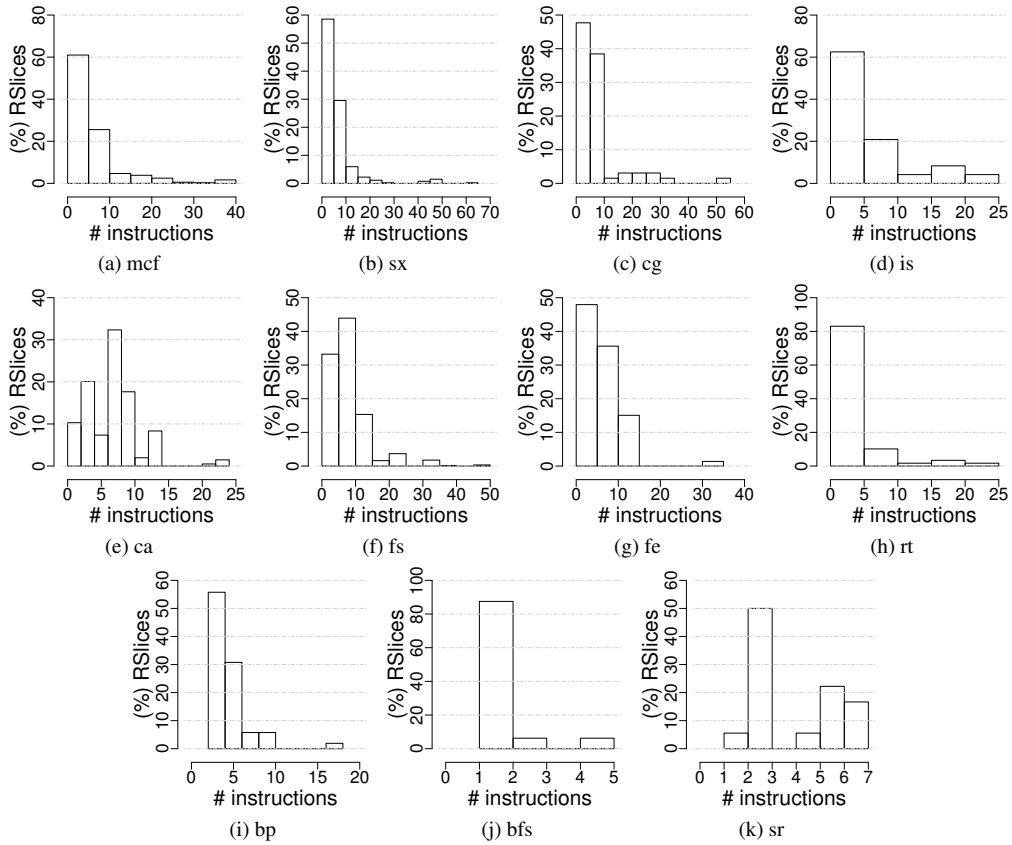

Figure 6: Histograms of instruction count per recomputed *RSlice* under *Compiler* policy.

along *RSlice(v)* instead. As the respective *RSlice(v)*s of sr are not as short, hence cheap, as the ones of bfs (Section 5.4), such excess recomputations cause *Compiler* to render a 7% degradation of EDP.

5.4 *RSlice* Characteristics

The number of instructions in an *RSlice* (i.e., *RSlice* length) is a fundamental determinant of the cost of recomputation. As *RSlice* length increases, recomputation incurs a higher cost due to the (re)execution of a larger number of instructions. Recomputation, i.e., traversal of an *RSlice(v)* under amnesic execution, provides higher energy efficiency benefits if the target data value v resides in lower levels of the memory hierarchy, and, at the same time, if the respective *RSlice(v)* is relatively short.

Fig. 6 shows histograms of instruction count per (recomputed) *RSlice* under *Compiler* policy. Recall that *Compiler* always triggers recomputation, independent of where v resides in the memory hierarchy. Therefore, Fig. 6 covers the profile for the *entire* set of *RSlice*s (as identified by the amnesic compiler; Section 3.1). Overall, we observe that 78.32% of the *RSlice*s have a length less than 10 instructions, across the board. Only 0.09% of the *RSlice*s contain more than 50 instructions. According to the storage complexity analysis from Section 3.4, this implies a small footprint for SFile and IBuff (Fig. 2), which grow with *RSlice* length.

For example, for the is benchmark from NAS, more than 30% of the loads swapped for recomputation have their data residing in the main memory (Table 5). At the same time, as Fig. 6d reveals, the application features mostly short *RSlice*s. As a result, amnesic execution results in very high EDP gain (87% according to Fig. 3). Although bfs features much shorter *RSlice*s than is (Fig. 6j), its EDP gain remains significantly lower (18.54% according to Fig. 3), because 98.43% of its loads swapped for recomputation have their data residing in L1 (Table 5).

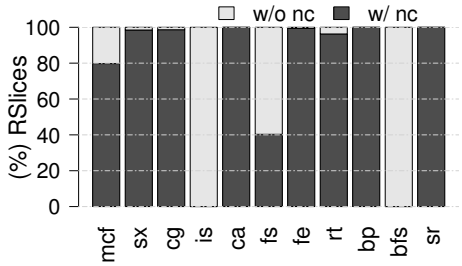

Figure 7: % of *RSlice*s with **n**on-re**c**omputable leaf inputs.

Hist from Fig. 2 only serves buffering non-recomputable (nc) leaf input operands of *RSlice*s. Fig. 7 shows the percentage share of *RSlice*s featuring non-recomputable leaf input operands for all applications. With the exception of is and bfs, such *RSlice*s represent the vast majority, rendering Hist a critical structure. According to our analysis, across all benchmarks, Hist has to record the non-recomputable in-

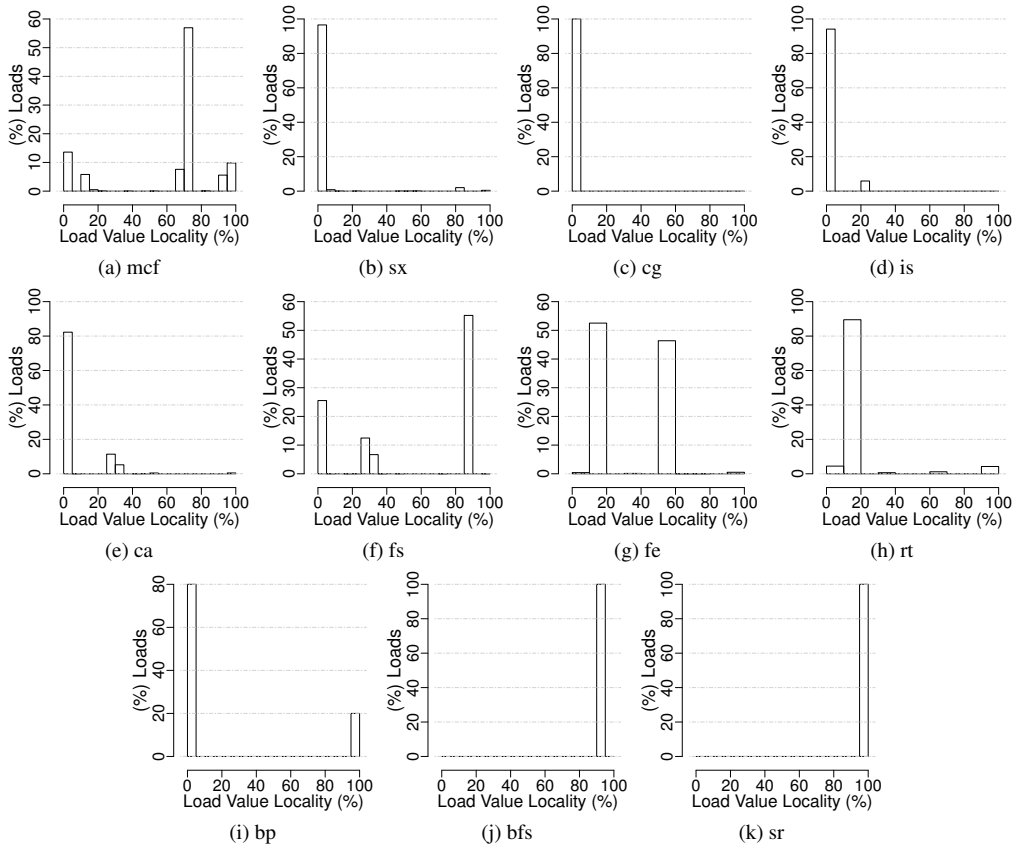

Figure 8: % value locality of loads (under classic execution), which are swapped for recomputation by the *Compiler* policy.

puts of at most 565 of such *RSlice*s at a time (i.e., for *fs*), where the average number of leaves is 1. A Hist design of no more than 600 entries can accommodate such demand (Section 3.4).

In the evaluation, we sized the microarchitectural components of Fig. 2 conservatively for the worst-case, to be able to capture the impact of recomputation without any bias. However, as Fig. 6 reveals, less than 50 entries for SFile or IBuff can cover most of the *RSlice*s. In this case, recomputation along excessively long *RSlice*s will not be possible, but long *RSlice*s are unlikely to deliver noticeable gains due to the higher (recomputation) cost incurred. Hence, we expect the gains from Fig. 3 mostly hold under practical sizing considerations.

5.5 Break-even Point

The basic idea behind amnesic execution is to swap energy-hungry load instructions with a sequence of non-memory (*Non-mem*) instructions to generate the respective data values. Each such sequence forms an *RSlice*. The non-memory instructions in an *RSlice* are mostly arithmetic/logic, as *RSlice*s do not feature memory or control flow instructions by construction (Section 3.1.1). The effectiveness of amnesic execution hence comes from such non-memory instructions being significantly less energy-hungry than load instructions, in today's machines at least.

The energy efficiency gain under amnesic execution tightly depends on the relative energy cost of non-memory instructions with respect to loads, i.e.,

$$R = EPI_{Non-mem}/EPI_{ld}$$

where $EPI_{Non-mem}$ captures the average EPI of a non-memory (i.e., arithmetic/logic) instruction; EPI_{ld}, of a load. R is a strong function of the underlying (micro)architecture and technology. The default value of R we used throughout the evaluation is

$$\begin{aligned} R_{default} &= EPI_{Non-mem,default}/EPI_{ld,default} \\ &= 0.45nJ/52.14nJ \approx 0.0086 \end{aligned}$$

which comes from the measured EPI estimates from [33] (Section 4). We next extract the value of R which would render amnesic execution useless, i.e., which would result in the same EDP under amnesic and classic execution. In other words, we analyze by how much the relative energy cost of non-memory instructions should increase (with respect to loads) to reach the break-even point for amnesic execution.

Bench.	$\mathbf{R_{breakeven}}$ (normalized)	Bench.	$\mathbf{R_{breakeven}}$ (normalized)
mcf	66.74	fe	13.7
sx	53	rt	45.63
cg	22.89	bp	83.25
is	73.74	bfs	3.89
ca	30.71	sr	36.74
fs	32.35		

Table 6: Break-even point (for *C-Oracle*).

As the relative energy cost, R, increases, amnesic execution becomes less and less beneficial, and past the value of R

at the break-even point, $R_{breakeven}$, as expensive as classic execution. Table 6 lists $R_{breakeven}$, normalized to $R_{default}$, for all of the benchmark applications. Each benchmark application reaches the break-even point at a different value of R due to the differences in the instruction mix (and hence, in *RSlice*s). For example, for bfs to reach the breakeven point, R (the relative cost of a non-memory instruction with respect to a load) should increase by 3.89× over its default, $R_{default}$. $R_{breakeven}/R_{default}$ takes much higher values for the rest of the benchmarks. In conclusion, unless R increases over $R_{default}$ by the coefficients provided in Table 6, amnesic execution is likely to stay more energy-efficient than its classic counterpart. Considering current technology projections [14], such increases are unlikely.

5.6 Data Locality Analysis

Fig. 8 shows the % value locality of load instructions, which are swapped for recomputation under the *Compiler* policy. In other words, these are the loads which get replaced by *RSlice*s. Without loss of generality, we stick to the *Compiler* policy in order to cover the entire set of swapped loads – recall that *FLC* and *LLC* policies only selectively swap loads for recomputation, while *Compiler* always enforces the swap.

We observe that, except bfs and sr, all of the benchmarks exhibit relatively low value locality for the swapped loads – the percentage of the swapped loads that have higher than 95% value locality remains less than 28% across the board. For bfs and sr, all of the swapped loads exhibit around 90% (Fig. 8j) and 99% (Fig. 8k) value locality, respectively. For cg, value locality is practically 0% (Fig. 8c).

This analysis indicates that amnesic execution is mostly *orthogonal* to alternative approaches such as load value prediction [24, 26] or memoization which exploit value locality to mitigate communication overhead. Memoization represents the dual of recomputation: the idea is replacing frequent and expensive computation with table look-ups for pre-computed data. In this manner, memoization can mitigate the communication overhead, since table look-ups are much cheaper than long-distance data retrieval. However, memoization is only effective if the data values generated by the respective computations exhibit significant value locality – in our context, these computations correspond to recomputation along *RSlice(v)*s to generate the data values v, and we capture in Fig. 8 the locality of such v by the value locality of the respective loads to read v from memory, without loss of generality.

6. Related Work

Kandemir et al. proposed recomputation to reduce off-chip memory area in embedded processors [16]. Koc et al. investigated how recomputation of data residing in memory banks in low-power states can reduce the energy consumption [20], and devised compiler optimizations for scratchpads [19]. These compiler strategies are limited to array variables. Amnesic execution is not necessarily confined to static compiler analysis or specific data structures. At the same time, as opposed to amnesic execution, these studies fail short of exploring opportunities for hardware-software codesign. **DataScalar** [5] trades computation for communication by replicating the most frequently accessed pages in each processor's local memory in a distributed system. As opposed to DataScalar, amnesic execution leverages recomputation at a much finer microarchitectural granularity. **Near memory processing (NMP)** [35, 22, 30, 17, 29, 21, 31] can bridge the gap between logic and memory efficiencies by embedding computation capability in main memory. Similar to amnesic execution, NMP can minimize energy-hungry data transfers. Amnesic execution and NMP are orthogonal, and NMP can benefit from amnesic execution to boost energy efficiency, or to reduce the memory footprint. **Memoization** [34, 12], the dual of recomputation, replaces (mainly frequent and expensive) computation with table look-ups for pre-computed data. Similar to NMP and amnesic execution, memoization can mitigate the communication overhead, since table look-ups are much cheaper than long-distance data retrieval. Memoization is only effective if the respective computations exhibit significant value locality. Therefore, memoization and recomputation can complement each other in boosting energy efficiency. **Idempotent Processors** [10] execute programs as a sequence of compiler-constructed idempotent (i.e., re-executable without any side effects) code regions. *RSlice*s aren't required to be strictly idempotent, but idempotent regions can act as *RSlice*s. Variants of **Speculative Precomputation** [37, 9, 32, 27, 36, 6] rely on speculative helper threads which run along main threads of execution to enhance performance (by e.g., masking long latency loads from main memory). Prefetching by helper threads can result in notable performance boost, however, helper threads still perform costly (main) memory accesses. The redundancy in execution incurs a power overhead on top.

7. Conclusion

In this paper, we investigate the effectiveness of recomputing data values in minimizing, if not eliminating, the overhead of expensive off-chip memory accesses. The idea is replacing a load with a sequence of instructions to recompute the respective data value, only if it is more energy-efficient. We call the resulting execution model *amnesic*. We detail an illustrative proof-of-concept design, identify practical limitations, and provide design guidelines. The proof-of-concept implementation features an amnesic compiler, microarchitectural support for amnesic execution, and an instruction scheduler to orchestrate amnesic execution at runtime.

Overall, we find that amnesic execution can reduce energy-delay-product of sequential execution by up to 87%, 24.92% on average, for 11 out of 33 benchmarks deployed. The rest of the benchmarks did not benefit much from recomputation (only 4 provided more than 5% gain), mainly because they did not feature many energy-hungry loads.

References

[1] ABRAHAM, S. G., SUGUMAR, R. A., WINDHEISER, D., RAU, B. R., AND GUPTA, R. Predictability of Load/Store Instruction Latencies. In *International Symposium on Microarchitecture (MICRO)* (1993).

[2] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Conference on Supercomputing (SC)* (1991).

[3] BERGMAN, K., BORKAR, S., CAMPBELL, D., CARLSON, W., DALLY, W., DENNEAU, M., FRANZON, P., HARROD, W., HILLER, J., AND KARP, S. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. *DARPA Information Processing Techniques Office (IPTO) sponsored study* (2008).

[4] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Tech. Rep. TR-811-08, Princeton University, 2008.

[5] BURGER, D., KAXIRAS, S., AND GOODMAN, J. R. Datascalar Architectures. In *International Symposium on Computer Architecture (ISCA)* (1997).

[6] CARLSON, T. E., HEIRMAN, W., ALLAM, O., KAXIRAS, S., AND EECKHOUT, L. The Load Slice Core Microarchitecture. In *International Symposium on Computer Architecture (ISCA)* (2015).

[7] CARLSON, T. E., HEIRMAN, W., AND EECKHOUT, L. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis* (2011).

[8] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization* (2009).

[9] COLLINS, J. D., WANG, H., TULLSEN, D. M., HUGHES, C., LEE, Y.-F., LAVERY, D., AND SHEN, J. P. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *International Symposium on Computer Architecture (ISCA)* (2001).

[10] DE KRUIJF, M., AND SANKARALINGAM, K. Idempotent Processor Architecture. In *International Symposium on Microarchitecture (MICRO)* (2011).

[11] GONZALEZ, R., AND HOROWITZ, M. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits 31*, 9 (1996).

[12] GUO, X., IPEK, E., AND SOYATA, T. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)* (2010).

[13] HENNING, J. L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News 34*, 4 (2006).

[14] HOROWITZ, M. Computing's Energy Problem (and what we can do about it). *Keynote at International Conference on Solid State Circuits* (2014).

[15] HU, Z., KAXIRAS, S., AND MARTONOSI, M. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *International Symposium on Computer Architecture (ISCA)* (2002).

[16] KANDEMIR, M., LI, F., CHEN, G., CHEN, G., AND OZTURK, O. Studying Storage-Recomputation Tradeoffs in Memory-Constrained Embedded Processing. In *Design, Automation and Test in Europe (DATE)* (2005).

[17] KANG, Y., HUANG, W., YOO, S.-M., KEEN, D., GE, Z., LAM, V., PATTNAIK, P., AND TORRELLAS, J. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design (ICCD)* (1999).

[18] KECKLER, S. W., DALLY, W. J., KHAILANY, B., GARLAND, M., AND GLASCO, D. GPUs and the Future of Parallel Computing. *IEEE Micro 31*, 5 (2011).

[19] KOC, H., KANDEMIR, M., ERCANLI, E., AND OZTURK, O. Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-processors. In *Design Automation Conference (DAC)* (2007).

[20] KOC, H., OZTURK, O., KANDEMIR, M., AND ERCANLI, E. Minimizing Energy Consumption of Banked Memories Using Data Recomputation. In *International Symposium on Low Power Electronics and Design (ISLPED)* (2006).

[21] KOGGE, P., BASS, S., BROCKMAN, J., CHEN, D., AND SHA, E. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Frontiers of Massively Parallel Computing* (1996).

[22] KOGGE, P. M. The EXECUBE Approach to Massively Parallel Processing. In *International Conference on Parallel Processing (ICPP)* (1994).

[23] LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)* (2009).

[24] LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1996).

[25] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)* (2005).

[26] MIGUEL, J. S., BADR, M., AND JERGER, N. E. Load Value Approximation. In *International Symposium on Microarchitecture (MICRO)* (2014).

[27] MOSHOVOS, A., PNEVMATIKATOS, D. N., AND BANIASADI, A. Slice-processors: An Implementation of Operation-based Prediction. In *International Conference on Supercomputing (ICS)* (2001).

[28] MOWRY, T. C., LAM, M. S., AND GUPTA, A. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (1992).

[29] OSKIN, M., CHONG, F., AND SHERWOOD, T. Active Pages: a Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture (ISCA)* (1998).

[30] PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. A Case for Intelligent RAM. *IEEE Micro 17*, 2 (1997).

[31] RIXNER, S., DALLY, W., KAPASI, U., KHAILANY, B., LOPEZ-LAGUNAS, A., MATTSON, P., AND OWENS, J. A Bandwidth-efficient Architecture for Media Processing. In *International Symposium on Microarchitecture (MICRO)* (1998).

[32] ROTH, A., AND SOHI, G. S. A quantitative framework for automated pre-execution thread selection. In *International Symposium on Microarchitecture (MICRO)* (2002).

[33] SHAO, Y., AND BROOKS, D. Energy Characterization and Instruction-Level Energy Model of Intel's Xeon Phi Processor. In *International Symposium on Low Power Electronics and Design (ISLPED)* (2013).

[34] SODANI, A., AND SOHI, G. S. Dynamic Instruction Reuse. In *International Symposium on Computer Architecture (ISCA)* (1997).

[35] STONE, H. S. A Logic-in-Memory Computer. *IEEE Transactions on Computers C-19*, 1 (1970).

[36] SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBURG, E. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000).

[37] ZILLES, C., AND SOHI, G. Execution-based Prediction Using Speculative Slices. In *International Symposium on Computer Architecture (ISCA)* (2001).