Trading Computation for Communication: A Taxonomy of Data Recomputation Techniques

Ismail Akturk, Ulya R. Karpuzcu

Abstract—A critical challenge for modern system design is meeting the overwhelming performance, storage, and communication bandwidth demand of emerging applications within a tightly bound power budget. As both the time and power, hence the energy, spent in data communication by far exceeds the energy spent in actual data generation (i.e., computation), (re)computing data can easily become cheaper than storing and retrieving (pre)computed data. Therefore, trading computation for communication can improve energy efficiency by minimizing the energy overhead incurred by data storage, retrieval, and communication. This paper provides a taxonomy for the computation vs. communication trade-off accompanied by a quantitative characterization.

Index Terms—data recomputation; communication reduction; energy efficiency; amnesic execution; load value prediction.

1 INTRODUCTION

ADressing the energy problem of modern computing [1] is not possible without understanding where the power goes. Figure 1 demonstrates a generic template for the sequence of events accompanying each step of classic computing: Upon retrieval of the input operands from the memory hierarchy (1 & 2), compute resources (be it general-purpose cores or specialized accelerators) derive the output data from the inputs (3), followed by storage (4 & 5) and retention (6) of the output data until the next update. Power goes to all of these events. The building blocks of classic processors, digital switches, consume dynamic power as they toggle, and – being far from ideal due to aggressive miniaturization – static power due to leakage when turned off.

Both the breakdown of total power consumption across events and the ratio of dynamic to static power per event evolve as a function of the operating regime and technology. Unfortunately, emerging technology solutions are not mature enough to meet the growing performance, storage capacity, and communication bandwidth demand within the tightly bound power budget (mainly due to cooling and power delivery limitations). At the same time, imbalances between logic and memory technologies cause energy (time \times power) consumption of data loads and stores (1, 2, 4 and 5) to significantly exceed the energy consumption of actual computation (3) [1], [2]. As a consequence, reproducing, i.e., recomputing data can become more energy efficient than storing and retrieving pre-computed data. This discrepancy is expected to become even more prevalent with technology scaling [3].

Figure 2(a) summarizes the classic trajectory at each step of execution from Figure 1. Black arrows point to the direction of data flow. Figure 2(b), on the other hand, captures how the picture changes by adapting recomputation. The idea is swapping the load from step 1 for the reproduction of the actual data values (which would otherwise be loaded from memory). Recomputation can reproduce such data values by brute-force recalculation [4], value prediction [5], [6], or approximation [7], [8] – spanning a three-way taxonomy. 1 incurs the time and power overhead of the memory access to perform the load; 2, of the subsequent communication of.

1. While Amnesiac [4] refers to recomputation as recalculation, we use recomputation in much broader sense in this paper: to refer to not only recalculation, but also prediction and approximation.
the respective data values; i.e., inputs to compute resources. Recomputation transforms the overhead of 1 & 2 to the overhead of the reproduction of the respective data values, which is similar to the overhead of 3. Therefore, recomputation can only improve energy efficiency if the cost of data reproduction remains less than the overhead of 1 & 2. In other words, the overhead of 1 & 2 sets the budget for recomputation. Under recomputation, the workload becomes more compute-intensive to make a better use of classic processors optimized for compute performance, as opposed to energy efficiency.

We will next look closer into the 3-way taxonomy of data recomputation techniques, accompanied by a quantitative compare and contrast. In the following, Section 2 covers the motivation; Section 3 introduces the taxonomy; Sections 4 and 5 provide the evaluation; Section 6 discusses related work, and Section 7 summarizes our findings.

2 Motivation

While emerging technology solutions alter the breakdown of total energy among stages 1 – 6 per Figure 1, the share of data communication and memory energy (i.e., 1, 2, 4, 5, 6) is projected to be predominantly higher [2]. At the same time, the inevitable quest for higher degrees of parallelism hurts data locality, therefore, increases communication energy further [1, 2].

2.1 Impact of Operating Regime & Process Technology

A promising way to boost energy efficiency is reducing the operating voltage, $V_{dd}$, aggressively to reach the switching threshold [10]. As $V_{dd}$ decreases, both dynamic and static power reduce, however, not at the same pace: Static power reduces less, and the share of static power grows. Figure 3 depicts how the share of static power (y-axis) evolves with technology scaling (x-axis). Each trend-line corresponds to a different $V_{dd}$. For each technology generation, the share of static power is optimized not to exceed 20\% under nominal conditions, when operating at nominal $V_{dd}$, $V_{ddNOM}$. For each technology generation, as $V_{dd}$ decreases from its nominal value $V_{ddNOM}$ by $0.75\times$, $0.5\times$, and $0.4\times$, the share of static power quickly increases. For example, at 22nm, as $V_{ddNOM}$ decreases by $0.4\times$, the share of static power approaches 50\%. Aggravated by shrinking feature sizes, variability in design parameters intensifies this effect: Due to variability, for any given $V_{dd}$, the share of static power tends to increase over technology generations. On the other hand, the impact of variability increases with decreasing $V_{dd}$.

As $V_{dd}$ decreases, the total power consumption reduces, however, a progressively increasing fraction of this reduced consumption goes to static power. Thus, static-power-heavy stages (mainly 6) become relatively more power hungry than dynamic-power-heavy phases (mainly 3). Emerging non-volatile memory technologies such as PCM [11, 12] or STT-RAM [13, 14] can minimize data retention (6) power due to practically zero static power, but suffer from excessive write (5) energy [15]. Recomputation can still be beneficial in this case, since recomputation can help reduce all components of data communication and memory energy (i.e., 1, 2, 4, 5, 6), including writes and data retention.

While energy efficiency assumes its maximum in the vicinity of approximately $0.4\times$ or $0.5\times V_{ddNOM}$ [15], operation at such ultra-low $V_{dd}$ can easily hurt data locality. This is because only increasing concurrency can prevent performance degradation due to the sizable drop in operating speed (frequency) at low $V_{dd}$ [10]. As a result, each core tends to spend both more time and power, therefore more energy, in data communication (i.e., 2, 4). Figure 3 does not consider this effect, which we will look closer into in Section 2.2.

3D stacking [16] or emerging photonics based interconnects [17] can render a lower data communication energy when compared to the state of the art, but would not alter the communication-centric nature of parallel processing; Orthogonal to the technology of the communication medium, higher levels of concurrency tend to hurt data locality, hence reduce the mean time to data communication. Accordingly, data communication is expected to remain as one of the most energy-hungry stages.

2.2 Concurrency vs. Data Locality

In a classic processor, cores communicate over the shared memory. Therefore, core-to-core communication translates into a sequence of core-to-memory (6) in Figure 1) and memory-to-core (2) in Figure 1) communication. The magnitude and the frequency of data exchange depends on the data distribution among the cores.

With increasing number of cores, the problem can distribute data to cores following strong or weak [18] scaling. Under both scaling paradigms, $n\times$ more cores increase the throughput performance by $n\times$ in the best case, if we exclude the overhead of communication. Table 1 captures how the total and per core problem size $PS$, execution time $t$, and throughput performance $PS/t$ evolve for an $n$-fold increase in core count.

<table>
<thead>
<tr>
<th>Scaling</th>
<th>(Total) $PS$</th>
<th>$PS$ per core</th>
<th>time ($t$)</th>
<th>$PS/t$</th>
<th>$PS$ share (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>const.</td>
<td>/n</td>
<td>/n</td>
<td>PS/t</td>
<td>/n</td>
</tr>
<tr>
<td>Weak</td>
<td>$\times n$</td>
<td>const.</td>
<td>$\times n$</td>
<td>/n</td>
<td>/n</td>
</tr>
</tbody>
</table>

Under strong (weak) scaling, overall $PS$ remains constant (increases by $n\times$), hence, $PS$ per core decreases by $n\times$ (remains constant). $t$ is proportional to $PS$ per core. As a result, $PS/t$ increases by $n\times$. At the same time, $PS$ share per core decreases, as tabulated in the last column, which represents the ratio of $PS$ per core (column 3) over the total $PS$ (column 2).
Recalculation Slice (RSlice)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

PS share serves as a proxy for data locality. This is because the total PS governs the total amount of data to be processed. Accordingly, PS share reflects the fraction of data processed by each core, which has to reside in close physical proximity to the core. Independent of the scaling paradigm, higher n – higher degrees of parallelism – tends to hurt data locality, hence, increase the likelihood of core-to-core communication. Factoring in the resulting data exchange overhead can easily wipe out the n-fold performance improvement. Indeed, time and power spent in data movement and the orchestration thereof is expected to dominate time and power spent in computation [1], [2].

3 Recomputation Taxonomy

The energy overhead of the load from Figure 2(a) determines the energy budget for recomputation. Unless the energy cost of reproducing data remains less than the energy cost of the respective load, recomputation cannot improve energy efficiency. Therefore, whether recomputation can improve energy efficiency or not tightly depends on where the data reside in the memory hierarchy – it is the location of the data in the memory hierarchy which determines the energy cost of the load. On the other hand, recomputation also incurs an energy cost due to the introduction of recomputing instructions to reproduce the respective data values.

Recomputation can reproduce such data values by brute-force recalculation [4], value prediction [5], [6], or approximation [7], [8], which gives rise to a 3-way taxonomy:

- Under brute-force recalculation, the recomputation effort goes to the derivation of data values, by re-executing the producer instructions (of the data values, which would otherwise be loaded from memory).
- Under prediction, the recomputation effort goes to the estimation of data values by exploiting value locality – the likelihood of the recurrence of data values [6] within the course of execution.
- Under approximation, the recomputation effort goes to the reproduction of the data values, however, at reduced accuracy. Generally, these techniques come in two flavors: (i) approximate recalculation, and (ii) approximate prediction. In (i), the recomputation effort goes to the actual calculation of data values – as it is the case for brute-force recalculation, however, at reduced accuracy. In this case, the compute resources may only partially execute the producer instructions (e.g., by dropping a subset), or perform recomputation at reduced precision. In (ii), the recomputation effort goes to to the estimation of data values by exploiting value locality – as it is the case for prediction, however, approximately.

Be it recalculation or prediction based, depending on the accuracy of approximate reproduction of the data values, approximation may degrade the accuracy of the end results. In this study we evaluate recomputation techniques at iso-accuracy, moreover, without compromising accuracy. Hence, our analysis spans (full-accuracy) recalculation and prediction, and leaves approximation based recomputation to future work.

3.1 Recalculation Based Recomputation

Recalculation can be implemented in various ways. In the following, we will use a compiler-based proof-of-concept implementation similar to [4]: During code generation, the compiler replaces each energy-hungry load instruction with the sequence of (arithmetic/logic) producer instructions of the respective data values. To this end, the compiler recursively traces data dependencies. The sequence of producer instructions forms a backward slice, as depicted in Figure 4, which we will refer to as a Recalculation Slice or RSlice.

![Recalculation Slice (RSlice)](image)

Figure 4: Example Recalculation Slice (RSlice)

The compiler swaps a load with its respective RSlice only if recalculation of the corresponding data value along the RSlice is more energy efficient than performing the load. To cross-validate the accuracy of the proof-of-concept optimizing compiler pass adapted from [4], we implement an Integer Linear Programming (ILP) based mathematical formulation (provided in the Appendix), which solves recalculation-enabled energy minimization as an optimization problem. We use Basic Block (BB) as the recalculation granularity (instead of an instruction) for the ILP formulation. A BB represents a super-instruction with a bounded number of input and output values. A BB cannot incorporate a branch or jump, by definition. For the ILP formulation, a finer granularity (i.e., instruction) incurs a higher overhead for dependency tracking from the computational complexity point of view. On the other hand, a coarser granularity increases the recalculation cost (due to a higher number of instructions to be re-executed per eliminated load) and yields a more pessimistic solution. Basic block granularity provides a sweet spot in terms of computational complexity and accuracy for the ILP formulation. This ILP formulation suits itself well to compiler integration, as well.

3.2 Prediction Based Recomputation

Under prediction, the recomputation effort goes to the estimation of data values, instead of brute-force recalculation. Accurate estimation is only possible if data values (which otherwise would be loaded from memory) exhibit high
value locality – i.e., a high likelihood of recurrence [6] within the course of execution. For example, if a data value exhibits excellent (100%) locality, just storing the value in a dedicated buffer and retrieving it from there may turn out to be more energy efficient than recalculating it (Section 3.1) or loading it from memory. Even if the value locality remains less than 100%, such buffered history of values can be used for prediction. Recent work has shown that emerging applications can oftentimes mask prediction incurred inaccuracy due to potential errors in estimation, as implied by the imperfect value locality [6].

Value retrieval from the history buffer constitutes the main cost of prediction. Under imperfect value locality, a prediction algorithm can help estimate the respective value by using the buffered history of previously observed values. In this case, the cost of executing the prediction algorithm should also be considered. Approximation aside, classic load value prediction features a repair mechanism to restore data values in case of a misprediction, as well. The overall cost of prediction including repair should fit into the recomputation budget, which in turn is set by the energy overhead of the respective load. Prediction based recomputation can only be beneficial if its energy cost remains less than the energy overhead of this load.

### 3.3 Recalculation + Prediction

**Prediction** based recomputation (Section 3.2) exploits locality of data values which would otherwise be loaded from memory. With respect to **recalculation** (Section 3.1), **prediction** targets the value to be produced by the root node of the RSlice. Input values of RSlice nodes may also exhibit significant value locality. Let us assume that such a node $n$ resides at level $l$, and it is not a leaf. In this case, predicting $n$’s inputs may turn out to be more energy efficient than re-executing $n$’s producers residing at level $l+1$ of the RSlice. Hence, combining **recalculation with prediction** (i.e., **recalculation + prediction**) can result in pruned RSlices to harvest even more energy efficiency. Recall that, if retrieving input data of leaves requires energy-hungry memory accesses, recalculating along the RSlice cannot be of any use. Each intermediate node of the RSlice subject to prediction becomes practically a leaf, as re-execution past such nodes would no longer be necessary.

**Recalculation + prediction** can prune RSlices, however, even under pure **recalculation** (Section 3.1). RSlices can never grow excessively: the energy overhead of the respective load determines the budget for recomputation. The cost of **recalculation** increases with the number of levels, i.e., height of the RSlice, and the number of nodes residing at each level. The re-execution of each node instruction incurs an energy cost. At most, as many nodes can be re-executed (i.e., can reside in the RSlice) as can be fit into the recomputation budget. And **recalculation** can only improve energy efficiency if the cost of re-execution along the RSlice remains less than the recomputation budget, which is set by the energy overhead of the respective load. In this manner, the energy overhead of the load prevents excessive growth of the RSlice. Under **recalculation + prediction**, the cost of re-execution along the RSlice along with the cost of selective prediction constitute the cumulative cost of recomputation.

### 4 Evaluation Setup

We experiment with benchmarks from the SPEC2006 [19], PARSEC [20], NAS [21], and Rodinia [22] suites, which span emerging application domains (Table 2). In the evaluation, we only analyze the benchmarks which harvest sizable energy efficiency gain under recomputation. The analyzed mix contains both compute- and memory-intensive sequential or single-threaded applications. We use Sniper [23] for microarchitectural simulation. The simulated microarchitecture is modeled after an in-order single-core Intel Xeon Phi-like processor without loss of generality, which features an operating frequency of 1.09GHz at 22nm, an L1 instruction cache of 32KB (4-way, LRU), an L1 data cache of 32KB (8-way, LRU, WB), and an L2 cache of 512KB (8-way, LRU, WB).

We profile the native binaries (conforming to classic execution, hence excluding recomputation) of the benchmarks on Sniper: We record (i) value locality of instructions at runtime (to be exploited by prediction based recomputation); (ii) cache statistics, i.e., hit and miss rates, at runtime (to derive the probabilistic energy cost model of the compiler pass).

The energy per instruction (EPI) estimates per load, store, and non-memory instructions come from measured Xeon Phi data from [24], which for memory instructions, provides separate EPI estimates for each level $L_i$ in the memory hierarchy: $EPI_{Li}$. Using these $EPI_{Li}$ and cache statistics from Sniper, we extract probabilistic EPI estimates for loads as follows: We derive $Pr_{Li}$, the probability of having the load serviced by level $L_i$, using hit and miss statistics of $Li$ from Sniper. Then, the sum of $Pr_{Li} \times EPI_{Li}$ over all levels $i$ in the memory hierarchy gives the probabilistic energy cost per load. Using this energy cost per load, and the EPIs for non-memory instructions, the compiler pass swaps a load with its respective RSlice only if recalculating of the corresponding data value along the RSlice incurs a lower energy cost than performing the load.

We implement the compiler pass from Section 3.1 in a Pin [25] based tool, which (by using the probabilistic energy cost model detailed above and by tracking data dependencies) swaps load instructions in the binary for the respective RSlices, only if recomputation incurs a lower energy consumption. At the same time, this tool adjusts the binary
under prediction and recalculation+prediction following Sections 3.2 and 3.3. To identify its maximum potential, we restrict prediction with the prediction of the values which would otherwise be loaded (i.e., be produced by RSlice roots under recalculation). Under recalculation+prediction, on the other hand, prediction can target any RSlice instruction but the root. We deploy Sniper integrated with McPAT [26] to run these annotated binaries in order to collect performance and energy statistics under recomputation.

5 Evaluation

We next quantify the energy efficiency under recomputation and analyze the implications for execution semantics.

5.1 Impact on Energy and Performance

Figure 5 compares the energy consumption under recalculation, prediction, and recalculation+prediction based recomputation. This analysis accounts for the overhead of recomputing producer instructions (along RSlices) under recalculation (Section 3.1), and history buffer accesses under prediction (Section 3.2). However, we assume that one history buffer access suffices for value prediction at 100% accuracy (i.e., we omit any potential overhead due to prediction algorithms or potential repair). For this experiment, we set the value locality threshold to enable prediction to 90%: prediction only applies to instructions which exhibit at least 90% value locality. Prediction targets only the values to be reproduced by root instructions of RSlices (all instructions along which are re-executed under recalculation). Under recalculation+prediction, on the other hand, prediction can target any RSlice instruction but the root (Section 3.3).

Figure 5 reports the energy gain with respect to native execution, which excludes recomputation. We observe that except bp, bfs, and sr, the energy gain under prediction is insignificant. This is because only a small number of instructions exhibit a higher value locality than 90%. Due to its wider applicability, recalculation unlocks higher energy gains, ranging from 5.06% to 67.43%, except sr. The recalculation cost for sr remains generally higher than the cost of the respective loads. An interesting observation is that bfs obtains lower energy gain under prediction and recalculation+prediction when compared to recalculation alone. The reason is that the RSlices of bfs are very short, rendering recalculation always cheaper than prediction. At the same time, our proof-of-concept implementation gives the priority to prediction if a value exceeds the locality threshold set for prediction (i.e., 90%) under recalculation+prediction: in other words, we omit recalculation for all values that exhibit a higher value locality than the threshold (90% in this case), even though recalculation turns out to be less energy hungry than the respective load.

Therefore, the energy gain under recalculation+prediction cannot exceed the gain under recalculation for bfs. Overall, the energy gain due to recalculation+prediction remains limited for the majority of the benchmarks. The reason is twofold: the benchmarks either do not have enough value locality to exploit prediction (e.g. mcf, sx, is, ca, fs, fe, and rt), or recalculation is too costly (e.g. sr).

Figure 6 reports the corresponding improvement in performance (i.e., execution time) with respect to native execution. Generally, a similar trend to energy gain applies, except that the performance gain under recalculation for sr becomes more pronounced when compared to the energy gain.

Fig. 5: Energy gain under recomputation.

Fig. 6: Performance improvement under recomputation.

Fig. 7: EDP gain under recomputation.

Fig. 8: EDP gain under prediction as a function of value locality threshold for prediction.

Fig. 9: EDP gain under recalculation+prediction as a function of value locality threshold for prediction.
Figure 7 summarizes the resulting gain in energy efficiency in terms of EDP (energy-delay product [27]), with respect to native execution. Overall, recalculation+prediction maximizes the EDP gain, and recalculation remains effective as well, except sr (as explained above). Prediction is beneficial for bfs, bfs, and sr only – recall that even this gain under prediction is optimistic as we neglect any algorithmic or potential repair incurred overhead. Finally, recalculation+prediction results in 11.8% to 92.2% EDP gain across all benchmarks.

We next assess the sensitivity of EDP gain to the value locality threshold for prediction. Figure 8 reports the EDP gain under prediction; Figure 9, under recalculation+prediction, as we sweep the threshold between 50% and 100%. Each bar per benchmark represents a different value locality threshold from this range to enable prediction. Generally, as the threshold increases, the number of values exhibiting at least that much locality reduces – therefore, a lower number of predictions can be performed, and both the energy and performance gains drop accordingly. Among the benchmarks, bfs exhibits the highest value locality, hence, it benefits most from prediction. bfs and sr, as well, benefit from prediction if the threshold remains lower than 100% – as very small number of loads swapped for RSlices feature 100% value locality for these benchmarks. On the other hand, fs and mcf harvest sizable EDP gain under prediction only if the threshold remains lower than 90% and 80%, respectively. The remaining benchmarks have a very small number of load instructions that exhibit ≥ 50% value locality, so only a negligible EDP gain applies under prediction (which already represents an upper limit for actual gains, as we neglect any algorithmic or repair related overhead). Therefore, recalculation+prediction can generally provide higher EDP gains when compared to prediction. As mentioned before, bfs has small RSlices, thus, the associated recalculation cost usually remains lower than the cost of prediction. Accordingly, bfs shows higher EDP gain for 100% threshold (at which a smaller number of values can be predicted, by definition, when compared to lower values of the threshold) under recalculation+prediction. Overall, we observe that
our findings from Figure 7 generally apply over this wider range of threshold values. We can conclude that recalculation has wider coverage for recomputation than prediction. Next, we investigate why this is the case.

5.2 Impact on Execution Semantics
As explained in Sections 3.2 and 3.3, in the context of recomputation, prediction serves two purposes:

(i) to predict the values which would otherwise be loaded from memory (and which correspond to the values to be reproduced by RSlice roots under pure recalculation) under prediction;

(ii) to predict the input values of intermediate (non-root) RSlice nodes under recalculation+prediction.

Prediction can eliminate re-execution along an entire RSlice if the values are already computed. Non-root nodes (except the root) exhibiting sufficient (input) value locality to render a smaller RSlice, which in turn becomes less energy costly to execute.

For prediction based recomputation to work, the respective instructions should exhibit sufficiently high value locality. Figure 10 reports a histogram of % share of RSlices having a given node count (x-axis). A lower threshold enables more predictions, hence more producer instructions can get pruned, and the node count shrinks more. We observe that prediction at a value locality threshold of 50% can reduce the node count of RSlices up to 56%.

6 Related Work
Kandemir et al. proposed recalculation to reduce off-chip memory area in embedded processors [28]. Koc et al. investigated how the recalculation of data residing in memory banks in low-power states can reduce the power consumption by preventing frequent switching of the corresponding memory banks to high-power states for data retrieval [29]. Koc et al. further devised recalculation-aware compiler optimizations for scratchpad memories [30]. The compiler strategies from [29] and [30] are confined to array variables. Amnesiac [4], on the other hand, replaces energy-hungry loads with a sequence of low-energy arithmetic/logic instructions to recalculate the respective data values, i.e., values which would otherwise be loaded from memory. The goal is saving energy. Amnesiac is not limited to embedded processors or specific data structures. Therefore we use a similar technique to Amnesiac as a more generic representative for brute-force recalculation throughout this paper.

Processing in/near memory (PIM/PNM) [31], [32], [33], [34], [35] can bridge the gap between logic and memory speeds by embedding compute capability in/near memory. Processing in memory can minimize energy-hungry data transfers, as well, and is orthogonal to recomputation. Memoization [14], [36] the dual of recomputation replaces (mainly frequent and expensive) computation with table look-ups for pre-computed data. Similar to processing in memory and recomputation, memoization can mitigate the communication overhead (as long as table look-ups remain cheaper than long-distance data retrieval). Memoization and recomputation can complement each other in boosting energy efficiency. Similar to memoization, computation reuse [37], [38], [39], [40] tries to reduce the amount of computation to be performed. The idea of computation reuse is based on the observation that data-intensive applications run over again and again with either identical or very similar inputs. In such cases, there is a considerable redundant computation (as the input remains almost the same with the previous input), and computation is needed only for the inputs that are changed (which are very limited). Although the main motivation behind computation reuse is to boost performance, it can also improve energy-efficiency as long as the input similarity check and copying the previously computed result remain cheaper than computing the result from scratch. Compared to computation reuse, recomputation is suitable for wider range of applications since it does not rely on input similarity (i.e., recomputation can be used in applications that exhibit no input similarity, as well).

The inefficiency of traditional CPU-centric processing motivated similar a body of work for large-scale data analytics, as well: near-data processing/in-situ analysis strategies that take computation to the storage (rather than data to the processor). These approaches can minimize larger scale energy-hungry data transfers and, being of the same spirit as PIM/PNM solutions, are orthogonal to recomputation. For example, for data sets that cannot fit into main memory, Cho et al. [41] proposed an active SSD architecture which supports basic data processing functions (such as...
filtering and aggregation) that can be performed directly on flash memory controllers, relying on the high SSD-internal bandwidth and the embedded CPU in the SSD controller to avoid costly data transfers. Tiwari et al., on the other hand, exploited the idle cycles of the SSD controller to process SSD-resident data [42]. Compared to Cho et al. [41], their approach does not require any hardware changes to the SSD controller. Gu et al. took the idea of SSD-based near-data processing further and provided a user-programmable framework that features high-level language support, dynamic load balancing, and multi-core support [45].

7 CONCLUSION
Recomputation can minimize, if not eliminate, the prevalent power and performance (hence, energy) overhead incurred by data storage, retrieval, and communication, thus, render more energy efficient execution. This paper provided a quantitative proof-of-concept analysis for the computation vs. communication trade-off, along with a taxonomy. Recomputation replaces data load(s) from memory with the reproduction of the respective data. Unless the energy cost of reproducing data remains less than the energy cost of retrieving it from memory, recomputation cannot improve energy efficiency.

In this study, we explored (interactions between) two broad classes of recomputation techniques: brute-force recalculation and prediction. Under recalculation, the recomputation effort goes to the derivation of the data values (which would otherwise be loaded from memory), by re-executing the producer instruction(s) of the eliminated load(s). Under prediction, the recomputation effort goes to the estimation of the data values by exploiting value locality – the likelihood of the recurrence of values (which would otherwise be loaded from memory) within the course of execution. We find that recalculation has wider coverage for recomputation than prediction, mainly because prediction cannot be effective under limited value locality as opposed to recalculation.
APPENDIX

Mathematical Formulation

We adopt an integer linear programming (ILP) based formulation for energy minimization under brute-force recalcula-
tion. Table 3 lists input and output parameters along with the objective.

| Optimizer Inputs | | Optimizer Outputs |
|------------------|------------------|
| E_{rd}^i | (average) energy per memory read | RD_i |
| E_{wr}^i | (average) energy per memory write | WR_i |
| E_{inst}^i | (average) energy per non-memory instruction | C_i |
| \#rd_i | number of memory read instructions in BB_i | RC_i |
| \#wr_i | number of memory write instructions in BB_i | |
| \#inst_i | number of non-memory instructions in BB_i | |
| \#consumers_i | total number of consumers of BB_i | |
| \#producer_i | total number of immediate producers of BB_i | |
| BB_i | number of basic blocks in dynamic control flow | |

Objective: \( E_{opt} = \min \{ \text{total energy to execute } BB_i \} \)

| TABLE 3: The lingua franca for ILP-based formulation. |
|------------------|------------------|------------------|
| **Objective** | **Constraints** |
| \( E_i \) | \( RD_i \) = \{ 1, if \( BB_i \) reads from memory \}
| | \{ 0, otherwise \} |
| \( E_i \) | \( WR_i \) = \{ 1, if \( BB_i \) writes to memory \}
| | \{ 0, otherwise \} |
| \( E_i \) | \( C_i \) = \{ 1, if \( BB_i \) is computed \}, \{ 0, otherwise \} |
| \( E_i \) | \( RC_i \) = \{ 1, if \( BB_i \) is recomputed \}, \{ 0, otherwise \} |

The next set of constraints stem from dependencies between dynamic basic blocks. If \( BB_i \) is computed (i.e., \( C_i = 1 \), \( BB_i \) may have either read its inputs from memory (i.e., \( RD_i = 1 \), or demanded recomputation of its inputs by its immediate producers (i.e., for each immediate producer \( BB_p \) of \( BB_i \), \( RC_p = 1 \)). Accordingly, \( C_i = (RD_i + RC_p) = 0 \) applies, where \( p = 1, 2, ..., \#producer_i \), points to the \( p^{th} \) immediate producer of \( BB_i \), \( RD_i \), indicates that \( BB_i \) read data produced by its \( p^{th} \) immediate producer from memory. \( RC_i \) indicates that \( BB_i \) got its input by having its \( p^{th} \) immediate producer recomputed. \( BB_i \)'s immediate producers have immediate producers themselves. These non-immediate producers of \( BB_i \) may get transitively recomputed to generate \( BB_i \)'s inputs, as \( BB_i \)'s immediate producers are recomputed. The recomputation of \( BB_i \)'s non-immediate producers gives rise to a further set of constraints in a recursive fashion. For each immediate producer \( BB_p \) of \( BB_i \), \( \sum_{p=1}^{\#producer_i} RC_p - (RD_{i,p} + RC_{pp}) = 0 \) applies, where \( pp = 1, 2, ..., \#producer_{p} \), points to the \( pp^{th} \) immediate producer of \( BB_p \). Finally, a basic block should not be recomputed, if all of its consumers read their input data from memory. In other words, the basic block can be recomputed only if at least one of its consumers does not read the input data from memory: \( RD_{i,p} + RC_p < 1 \) for all immediate producers \( BB_p \) of \( BB_i \).

**References**
