RICH: Implementing Reductions in the Cache Hierarchy

Vladimir Dimić
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
vladimir.dimic@bsc.es

Miquel Moretó
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
miquel.moreto@bsc.es

Marc Casas
Barcelona Supercomputing Center
marc.casas@bsc.es

Jan Ciesko
Center for Computing Research,
Sandia National Laboratories
Albuquerque, NM, USA
jciesko@sandia.gov

Mateo Valero
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
mateo.valero@bsc.es

ABSTRACT

Reductions constitute a frequent algorithmic pattern in high-performance and scientific computing. Sophisticated techniques are needed to ensure their correct and scalable concurrent execution on modern processors. Reductions on large arrays represent the most demanding case where traditional approaches are not always applicable due to low performance scalability.

To address these challenges, we propose RICH, a runtime-assisted solution that relies on architectural and parallel programming model extensions. RICH updates the reduction variable directly in the cache hierarchy with the help of added in-cache functional units. Our programming model extensions fit with the most relevant parallel programming solutions for shared memory environments like OpenMP. RICH does not modify the ISA, which allows the use of algorithms with reductions from pre-compiled external libraries. Experiments show that our solution achieves the speedup of 1.11× on average, compared to the state-of-the-art hardware-based approaches, while it introduces 2.4% area and 3.8% power overhead.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**;
• **Computing methodologies** → *Parallel programming languages*;
• **Software and its engineering** → *Runtime environments.*

KEYWORDS

shared memory, caches, reductions, task-based programming model

ACM Reference Format:
Vladimir Dimić, Miquel Moretó, Marc Casas, Jan Ciesko, and Mateo Valero. 2020. RICH: Implementing Reductions in the Cache Hierarchy. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392736

ICS '20, June 29-July 2, 2020, Barcelona, Spain

1 INTRODUCTION

The CPU clock frequency stagnation due to the end of Dennard scaling [14] has forced hardware vendors to consider increasingly large core counts. To take advantage of these multi-core designs, the software needs to be able to run concurrently on many threads. Current high-performance systems are dominantly used in scientific domain, where applications often rely on operations like reductions. Therefore, the acceleration of such reductions is of paramount importance to optimize parallel executions and properly reach the best performance out of complex codes.

In general, reductions are operations that accumulate values on a given data structure, called reduction variable. The performance of a reduction operation is impacted by factors like the reduction variable size or the memory access pattern. Applying case-specific techniques is required to achieve optimal performance for the parallel execution of reductions. For example, in current systems, there are two main techniques used to parallelize reductions:

(i) Software privatization [8, 46] progressively accumulates the partial result into the thread-private copies of the reduction variable. Once the algorithm finishes, partial results in the private copies are combined into the final result. This solution works well with small reduction variables, i.e. scalar variables, structures or short arrays. However, in codes with large arrays, such as matrix-matrix multiplication, data replication introduces cache pollution that can significantly harm the application's performance.

(ii) As an alternative, programmers can use atomic operations [4, 25, 27, 41] for implementing reductions. This software solution outperforms privatization on large reduction variables. On the negative side, there is not always hardware support to atomically run all arithmetic and logic operations. Moreover, atomics often suffer from invalidations caused by the coherence protocol due to concurrent accesses to the same location as well as due to false sharing. Consequently, atomics usually perform worse than privatization-based software techniques for the case of reductions on scalars and small arrays, as we demonstrate in Section 2.1.

Several techniques using hardware extensions for reductions have been proposed, such as COUP [50] and PCLR [17], which implement hardware privatization of the reduction variable. Such solutions work well for small reduction variables. However, for reductions on larger arrays, cache pollution caused by copies of the reduction variable becomes an issue and may negatively affect performance. Moreover, these techniques modify the processor's

ISA, which makes them incompatible with external pre-compiled algorithmic and mathematical libraries commonly used in complex scientific and High Performance Computing (HPC) applications.

While there are well-performing solutions for reductions on scalar variables, vector reductions still remain an open topic of research. In order to solve the aforementioned challenges, we propose RICH, a runtime-assisted technique for performing reductions in the processor's cache hierarchy. The goal of RICH is to be a universally applicable solution regardless of the reduction variable type, size and access pattern. For its implementation, we design a hardware extension equipped with functional units to perform reductions at any level of the cache hierarchy. Existing constructs in a shared-memory parallel programming model are extended to let the programmer specify at which location in the cache hierarchy a certain reduction should be computed. The runtime system couples the application with the operating system, with the goal to provide the underlying hardware with the information about the reduction variable. This interface is designed without modifying the processor's ISA. As a result, RICH supports the use of algorithms with reductions implemented in third-party libraries.

This paper makes the following contributions:

- RICH enables the programmer to offload the reduction operation from the core to a desired level of the cache hierarchy. This functionality is facilitated by extending existing OpenMP-like annotations in the parallel code.
- We propose a new hardware component, the Reduction Module, able to perform reductions at all levels of the cache hierarchy.
- Our design couples the parallel application and the underlying hardware with a runtime-assisted interface that does not modify the processors ISA. As a result, RICH is applicable to common scenarios where complex codes use reduction algorithms implemented in third-party pre-compiled libraries, which is not supported in the state-of-the-art hardware techniques for reductions, such as COUP [50] and PCLR [17].
- Experimental results for vector-reductions show that RICH achieves performance improvements of 1.8× on average, compared to the current approaches implemented in parallel programming models. With scalar-reductions, RICH outperforms software privatization 1.09× on average. RICH performs on average 1.11× faster than COUP.

The rest of the document is structured as follows. Section 2 provides context for our work and discusses current approaches for performing reductions. Section 3 describes RICH, including the architectural support for reductions and compiler and programming model extensions. Section 4 explains the experimental methodology and introduces the benchmarks used for the evaluation of RICH. Section 5 discusses the most important design decisions, while Section 6 presents a detailed evaluation of the proposal. Section 7 describes the related work and Section 8 concludes this document.

2 BACKGROUND AND MOTIVATION

In the context of parallel programming, reductions are operations where input data is accumulated by applying an operator to generate output data [24]. For the rest of the article, we define a reduction variable as a data structure that holds the output data of the reduction. Addition and multiplication are commonly used as reduction operators in scientific computing. Reductions can be parallelized because these operations are associative and commutative [37]. Based on the reduction variable's size, we classify reductions into two categories:

(*i*) Reductions over scalar-types. This type of reductions occurs in a wide range of application domains including combinatorics (e.g. satisfiability problems such as n-Queens) or scientific computing (e.g. normalized residuals to verify convergence or code correctness) or to implement hardware performance counters. On current mainstream architectures, updates to shared data should be avoided, since they may result in high cache coherence traffic that significantly impacts execution performance.

(*ii*) Reductions over vector-types. Such reductions are usually present in more complex scientific codes that accumulate results on arrays or higher-dimensional matrices. Depending on how the reduction variable is accessed during the reduction, we identify two sub-categories: near-linear access patterns and irregular access patterns. Near-linear reductions typically take place in scientific codes where operations access just the neighboring grid elements, such as in LULESH [29] and SPECFEMD [31]. Irregular array-type reductions are frequently found in n-body codes, histogram computations, as well as in applications where data structure representing a physical domain is accessed in an irregular manner. Concurrent execution of vector-reductions requires solutions that avoid unnecessary data privatization, prevent data races due to concurrent updates to overlapping memory regions and effectively reduce memory bandwidth and latency requirements.

2.1 Software-Based Solutions for Reductions

There are two intuitive software-based techniques to parallelize reductions. The first approach, called *privatization* [8, 46], consists in having each of the threads involved in the parallel execution performing a partial reduction over its portion of input data. The partial reduced data is stored in private per-thread copies of the reduction variable. Partial results are combined in the final result after the parallel reduction tasks are completed. Privatization works well for reductions over scalars and small arrays. For larger reduction variables, however, privatized data increases cache pollution.

As an alternative, private threads directly update the shared reduction variable. To ensure correctness, the update operation is guarded using *atomic* instructions that are commonly implemented in modern processors [1, 4, 25, 27]. This method performs worse than privatization for small reduction variables due to frequent cache misses caused by the invalidations of cache lines in the private caches as well as the increased coherence traffic. Figure 1 shows an example of such behavior by comparing the achieved memory bandwidth of the two techniques mentioned above considering different reduction variable sizes. For this analysis we use RandomAccess [2], a kernel that accesses the reduction variable following a uniform probability distribution function.

Results show that for small array sizes, privatization is the approach delivering higher performance as it avoids the shared updates. In contrast, atomics achieve significantly lower performance for small problem sizes due to the coherence effects. The atomics performance improves when the array size increases as conflicts between different threads are less likely to occur. As the array size approaches the size of the core's portion of the shared L3 cache,

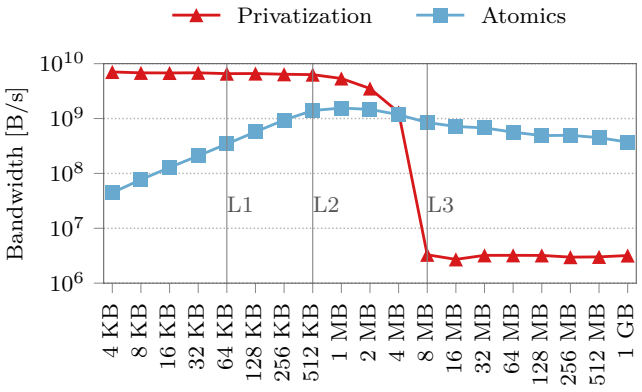

Figure 1: Achieved memory bandwidth for RandomAccess benchmark for different reduction array sizes on an IBM POWER8 processor with 192 threads. Vertical lines show the sizes of the data caches per core.

privatization suffers from a performance drop of 1000× due to the overhead of handling the private copies of the reduction variable. Although atomics also show a notable drop in performance, they perform significantly better than privatization. With the further increase of the reduction variable's size beyond 8 MB, the performance of privatization stagnates, while the performance of atomics slowly degrades.

This analysis clearly shows how different reduction methods deliver different performance depending on the size of the reduction variable. Specifically, the size of the reduction variable dictates which reduction technique should be used to achieve better performance. Solutions allowing a manual or automatic selection of the reduction technique are required in order to achieve the best possible performance across all possible scenarios without exposing the programmer to the complexity of implementing application-specific ad hoc reduction techniques. To further reduce the overheads of software techniques, hardware solutions are necessary.

2.2 Hardware-Assisted Reductions

There are several state-of-the-art hardware techniques addressing issues related to coherence invalidations or data privatization costs. They either implement atomic remote memory accesses [3] or use private cache lines [17, 50]. Remote atomic updates implement atomicity by performing the final reduction, involving all the partial results, at a specific hardware component. These components can be the last-level cache or the memory controller equipped with additional functional units. The use of private cache lines is based on the same concepts as its software privatization counterpart. In this case, processor caches are used as temporal buffers to accumulate intermediate results. Private cache lines are initialized to the neutral element on the first access and are reduced at cache line eviction or at the end of a software routine by generating the final value in the last-level cache. Described designs avoid the high coherence traffic triggered by shared updates to the reduction variable.

However, previous proposals do not perform optimally in all scenarios. Solutions based on remote memory accesses are not suitable for small reduction variables due to higher chance of conflicts and the resulting serialization of update operations. Applications with irregular memory accesses do not efficiently use cache memories because such access patterns exhibit low spatial and temporal locality. The usage of private cache lines in such codes results in a sequence of initialization, cache placement and eviction events. Moreover, in the case of large reduction arrays, privatizing the reduction variable significantly pollutes the content of cache memories. Consequently, further architectural innovations are needed to avoid these issues while keeping the benefits of low coherence traffic.

2.3 Ongoing Challenges

Reductions on scalar variables or small output arrays are well supported in current designs. However, reductions considering large arrays or displaying irregular access patterns require novel techniques to avoid performance degradation due to cache pollution and increased coherence traffic. Proposed hardware and software solutions are just suitable for a subset of scenarios, depending on the size of the reduction variable and its memory access pattern. To the best of our knowledge, a technique effective for all reduction scenarios has not been proposed. Moreover, previously proposed hardware techniques require ISA extensions to handle reduction operations, which makes them incompatible with applications that use pre-compiled libraries containing reduction operations.

In this article we propose a solution aimed at achieving the following goals: (i) To achieve better performance than the state of the art considering a wide range of reduction variable sizes and different memory access patterns. (ii) To avoid modifications to the processor's ISA and thus maintain the compatibility with pre-compiled and dynamically linked libraries. (iii) To let the programmer expose application-specific knowledge to the hardware without the need for ad hoc implementations of reductions.

3 RICH: IMPLEMENTING REDUCTIONS IN THE CACHE HIERARCHY

RICH is a runtime-assisted technique for performing reductions in the cache hierarchy. The programmer makes use of simple source code annotations to identify reduction variables and specify both the reduction operator and the hardware components where reductions should take place. Such annotations are expressed in terms of pragma directives [36]. The runtime system is responsible for providing the hardware with the information specified by the programmer. Finally, the additional hardware components in the processor's caches are responsible for handling and executing the reduction operations. RICH relies on the following extensions:

- Programming model support to define the reduction technique and the runtime system extensions to set up the relevant hardware components.
- A novel hardware component, called Reduction Module (RM), located at the cache hierarchy. The RM performs the reduction instructions issued by the cores.
- Microarchitectural extensions in the processor and its memory hierarchy to handle reduction requests in the core and their propagation to the RM through the cache hierarchy.

In this section we describe these extensions in detail. Finally, we discuss different design decisions and the implications of the proposed processor functionalities.

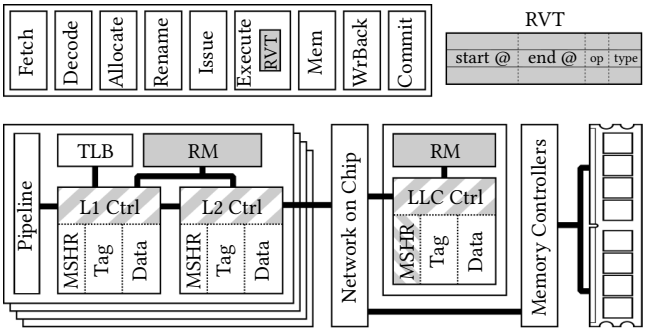

Figure 2: The pipeline schematic with the added RVT component (top left); Microarchitecture of the RVT (top right); Microarchitecture of the memory hierarchy with the added Reduction Modules (bottom). New components are colored in solid gray and modified components in striped gray.

3.1 Microarchitectural Support for Reductions

In the following paragraphs we describe the hardware modifications we propose to execute reduction operations in the cache hierarchy. Figure 2 shows the relevant details of a multi-core processor microarchitecture with the added (solid gray) and modified (striped gray) components. We describe our proposal in a context where each core is equipped with two levels of private caches while the Last-Level Cache (LLC) is shared among all cores. Our architectural innovations can be also deployed, with minor adaptations, in other contexts with different cache memory hierarchies. A Reduction Module (RM) is added to the private caches of each core as well as to the LLC. The private caches share a single RM. The cache controllers are modified to communicate with the RM and to handle reduction store instructions. A small hardware component that holds the range of reduction variables for the current thread is placed in each core. All added and modified hardware structures are described in detail in the following paragraphs.

Recognizing reduction instructions is partially facilitated by a special hardware structure called Reduction Variable Table (RVT). For a given address, the RVT determines if the address belongs to a reduction variable in the current thread. For load and store instructions within the reduction task, the RVT is accessed in the execute stage, once the destination address of the memory operation is calculated. The RVT holds the ranges of virtual addresses corresponding to the reduction variables (*start @* and *end @*), as well as the data type (*type*) and the operator (*op*) used for accumulating values into each reduction variable. The content of the RVT is managed by the runtime system, as explained in Section 3.2.

The reduction operation is composed of: a load from the reduction variable into a register, an arithmetic or logic operation that updates this register and a store of the modified register to the original memory location. The load and the store instructions are detected by a lookup of the load and the store addresses, respectively, in the RVT. A successful lookup to the RVT signals to the core that the address belongs to a reduction variable and that the corresponding instruction is a reduction instruction. The arithmetic operation that has the chain register dependency with the reduction load and store instructions is also designated as a reduction instruction. Such design does not require load-modify-store instructions to be consecutive.

Depending on the target architecture, the atomicity of the load-modify-store chain is achieved in different ways: (i) Load-Link and Store-Conditional instructions [4, 25, 41] and (ii) Compare-And-Swap construct [27]. We implement RICH to support both synchronization mechanisms. Since RICH uses only the address accessed by the loads and stores to determine if they participate in reduction operation, it is not important which mechanism is used to ensure the atomicity of the reduction operation.

RICH supports reduction operations that update the reduction variable with a sequence of load-modify-store instructions. All reduction operators defined in the OpenMP standard 5.0 have this property. This covers arithmetic instructions ADD, SUB, MUL, logical operations AND and OR, bitwise operations AND, OR and XOR and MIN/MAX. In addition, RICH supports the DIV operation. Operations on both integer and floating point data are allowed.

When a core recognizes a reduction operation, the arithmetic or logic instructions involved in it plus the load instructions to the reduction variable are converted into NOP instructions in the core's pipeline. After effectively eliminating these instructions, the CPU converts the reduction store instruction into a special store instruction that holds information from these removed instructions: the reduction operator, the data to be reduced and the reduction variable's address. The special store instruction is propagated through the cache hierarchy until it arrives to the cache level configured to perform the reduction. To ensure the correctness of this design, we do not permit any instruction consuming the reduction variable to execute before the reduction operation has finished. This is enforced by using existing OpenMP synchronization primitives such as barriers or dependencies between different user functions [36].

Since the load and the arithmetic or logic instruction involved in reduction are converted into NOP instructions, their destination registers will not hold the loaded or computed value. This does not present an issue due to the fact that programmer guarantees that the reduction task only updates the reduction variable and does not consume it. Therefore, an unmodified compiler already generates a code that does not consume the values stored in these registers. However, the compiler is allowed to reuse these registers for independent instructions to store another variable, even inside the reduction task. The mechanisms already present in the processor pipeline ensure the correctness and efficiency of such execution in an out-of-order processor.

Reduction Module. Figure 3 shows the microarchitecture of a Reduction Module (RM) which consists of the following three hardware structures:

The *RM Instruction Queue* (RMIQ) contains instructions that are to be executed or are being executed by the RM. The RMIQ is designed as a circular queue to maintain the order of the inserted instructions. Each entry in the RMIQ contains information specified by a reduction instruction, i.e the reduction operation to be performed (op), the address of the reduction variable (addr), its size (sz) and the value that is to be reduced into the reduction variable (val). The *data* field holds the current value of the accessed location within the reduction variable, which may not be available in cache at the time of inserting an instruction into the RMIQ. In

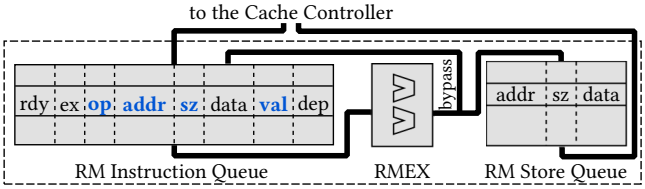

Figure 3: Microarchitecture of the Reduction Module.

that case, the entry is marked as "not ready" (field *rdy*). Only ready instructions can be executed. The *ex* field indicates whether the instruction is being executed. The *dep* field points to an entry in the RMIQ that depends on the result of this instruction.

The *RM Execution Unit* (RMEX) contains the logic that performs arithmetic and logic operations on all standard data types. It consists of an Arithmetic-Logic Unit (ALU) and a Floating-Point Unit (FPU).

The *RM Store Queue* (RMSQ) is a circular buffer for storing the results of the reductions until they can be written back to the cache's data storage. Entries of the RMSQ contain an address, the corresponding data and its size. Whenever a cache's write port is not in use, the controller writes the oldest entry from the RMSQ into the cache and removes it from the RMSQ.

RICH configurations. In our proposed architecture, the system can be configured to perform reductions at different levels of the cache hierarchy, i.e. at any of the private caches or the shared last-level cache. Although the inner behavior of the RM is the same, the handling of the reduction instructions in the cache controller depends on the RICH configuration. Depending on the cache level where the reduction is performed, we define the following three configurations $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$. Configurations $RICH_{L1}$ and $RICH_{L2}$ imply that partial reduction is performed in the corresponding private caches. After the reduction task is finished, the reduction lines from caches are written back and the final reduction is carried out at the LLC level. In the $RICH_{LLC}$ configuration, only the RM in the LLC is active. Inactive RMs do not consume energy as they are turned off by the power gating mechanism [38].

Processing a reduction instruction in caches. The cache controllers are set up to either process the special reduction store instructions or delegate it to the next cache in the memory hierarchy, depending on the selected RICH configuration. This setup is performed by the runtime system before the task user code starts executing. When a special store instruction involved in a reduction reaches the cache level where it will run, it is inserted in the RMIQ and marked as "not ready". Before the instruction can start executing, the current value of the reduction variable needs to be fetched into the RM. Depending on the state of the RM and the corresponding cache, different actions are performed:

- If the RMIQ contains an entry reducing to the same address as the new instruction, the new instruction needs to wait for the data from the preceding instruction, whose *dep* field is updated to point to the newly inserted instruction.
- Otherwise, the reduction data has to be read from the RMSQ or the data cache. If the RMSQ contains an entry matching the address of the new reduction instruction, data is read from the RMSQ into the *data* field in the RMIQ and the new instruction is marked as ready.
- If the reduction is performed in a private cache ($RICH_{L1}$ and $RICH_{L2}$), data is fetched from the cache in the case of a *cache hit*. In case of a *cache miss*, a cache line is allocated and filled with neutral elements corresponding to the reduction operator. When a cache line holding the reduction variable is evicted from a private cache, it is reduced by the RM inside the LLC.
- If the reduction is performed in the shared cache ($RICH_{LLC}$), the data is fetched from the cache's data storage. In case of a *cache miss*, a standard request is sent to the memory controller and the pointer to the RMIQ entry requesting the data is inserted into the MSHR. Once the data arrives to the cache, it is written into the appropriate entry in the RMIQ, simultaneously marking the entry as ready.
- In scenarios where the data is present in a level of the cache hierarchy lower than the level where the reduction takes place (e.g., the valid data is located in the L1 cache in $RICH_{L2}$ configuration), the unmodified coherence protocol moves the data to the desired cache level.

When scheduling an instruction for execution, the controller takes the first ready instruction from the head of the RMIQ, sets its *ex* bit and forwards the entry to the RMEX for execution. Once the execution finishes, the result, together with the destination address, is stored in the RMSQ, while the corresponding entry in the RMIQ is freed. In case an instruction is waiting in the RMIQ for the output of the finished reduction operation, this output is written in the *data* field of the corresponding entry, marking it as ready. Entries from the RMSQ are written back to the cache's data store when the cache's write port is available and removed from the RMSQ.

When a request is sent to the RM, the corresponding cache line is locked, which prevents it from being evicted. The lock guarantees that the line is present for the write-back operation from the RM, which releases the lock upon completion.

Accessing a reduction variable outside of the reduction scope. Once the reduction finishes, the application often accesses the reduction variable for further processing. It is necessary to differentiate between accesses generated inside the reduction scope and those accesses that happen outside of reduction context. RICH uses the RVT to recognize reduction instructions. The runtime system populates the RVT before the reduction context begins and clears it after the reduction is finished. This mechanism is described in Section 3.2. If the variable is accessed outside of the reduction context, the request is processed as a normal memory instruction. Also, we do not allow instructions accessing the reduction variable that do not belong to the reduction operation to run before the whole reduction has finished. This is automatically enforced by OpenMP synchronization primitives [36], which are inserted by the source-to-source compiler.

Memory consistency. Memory consistency of non-reduction data is not affected. All loads and stores are issued by the core in a way that maintains Total Store Order (TSO) memory consistency model [45]. On the other hand, the loads and stores issued by the RM and non-dependent, non-reduction loads and stores issued by the core can be seen in different order by the memory subsystem. To guarantee that an access to reduction variable never returns a wrong value, (i) the programmer ensures that, within the reduction task, the reduction variable is only accessed with read-modify-write

construct, i.e., reduction operation, and (ii) the source-to-source compiler inserts a memory fence after a reduction task to guarantee that successive consumer task accesses the correct data.

Cache coherence. RICH does not modify the cache coherence protocol. Depending on the RICH configuration, specific explicit synchronization actions are performed to guarantee coherence of reduction data in the caches. In all configurations, the reduction variable can either be present in the cache's data store or in the RMSQ of the same cache. The cache controller considers both locations when searching for a cache line of an in-flight reduction variable. $RICH_{L1}$ and $RICH_{L2}$ require a final reduction of the partially reduced data, which is performed at the end of the reduction task. A memory fence, inserted by the source-to-source compiler, guarantees that these data are not consumed before the final reduction takes place.

Support for precise exceptions and speculation. RICH implementation maintains support for precise exceptions by guaranteeing in-order retiring of instructions. Events caused by exceptions and mis-speculations are bidirectionally communicated between the core and the RM. In case of an exception or mis-speculation, the appropriate in-flight instructions in the RM are flushed, new values stored in the RMSQ are discarded and old values stored in the RMIQ are restored.

3.2 Programming Model and Compiler Support

The programming model support for the proposed hardware design relies on the existing implementations of the most popular shared memory parallel programming model, OpenMP [36]. OpenMP supports both loop-level and task-based parallelism. In task-based codes, the programming model offers explicit synchronization with *taskwait* constructs. In addition, if programmers define data dependencies between tasks, OpenMP automatically ensures correct execution in a data-flow manner by respecting the user-specified task data dependencies. Specifically, tasks that depend on data produced by other tasks are scheduled to execute only when these data dependencies are satisfied. When loop-based parallelism is employed, implicit barriers are added to enforce synchronization.

We design RICH to be agnostic to the applied parallelization technique. The proposed programming model extensions are built on top of the existing implementation of reductions in OpenMP. This is beneficial as any extension to a programming model requires careful design for consistency with minimal implications on unrelated constructs, user understanding and compatibility with previous versions and existing codes.

We extend the reduction directive as follows, with the added parameter shown in bold.

$$reduction(reduction\text{-}ident. : \textbf{\textit{[reduction-technique]}} : list)$$

As defined in Section 2.19.5.4 of the OpenMP 5.0 standard [36], *reduction-identifier* specifies the reduction operator while *list* specifies the list of reduction variables. The added optional field *reduction-technique* specifies which reduction technique to use (CPU, $RICH_{L1}$, $RICH_{L2}$ or $RICH_{LLC}$). The default configuration, CPU, executes the reduction operations in the core and does not use the hardware acceleration in the RM. Using the information specified in this

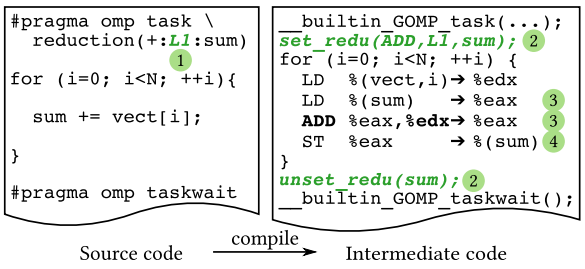

Figure 4: The code transformation done by the compiler.

annotation simplifies our design as it does not require adding special reduction instructions to the processor's ISA, and, therefore, maintains the compatibility with pre-compiled libraries.

The information specified in the programming model directives is forwarded to the RVT by a function call implemented in the runtime library using instructions on memory-mapped registers. This call is inserted by a source-to-source compiler in the code location where the executing thread encounters the beginning of the parallel region or a task that participates in a reduction. This source code location is considered as the start of the reduction scope, which is terminated once all tasks or iterations from that parallel region finish.

Figure 4 illustrates how a programmer uses the programming model extensions. The starting point in this example is a parallelized vector reduction code that uses the OpenMP reduction construct. The example is applicable to both task-based and loop-based parallel codes. The programmer selects the cache level where the reduction will take place by taking into account properties of the application like the workload size, reduction variable size, and its memory access pattern. In this example we decide to execute the reduction in the L1 cache, which is specified in the reduction clause 1, as defined earlier in this section. The source-to-source compiler inserts calls to functions implemented in the runtime system library used to populate the RVT with the information about reduction variables and the chosen reduction location 2.

During program's execution, the load and arithmetic operations belonging to reductions 3 are discarded. The store instruction 4 is enriched with the reduction operation type (`ADD`, from RVT) and the register holding the value to be reduced (`%edx`, from the preceding ALU instruction). The enriched store instruction is then forwarded to the core's RM. The further handling of reduction instructions by the hardware is explained in detail in Section 3.1.

4 EXPERIMENTAL METHODOLOGY

4.1 Benchmarks

In Section 2 we introduced the two categories of reduction operations based on the reduction variable size: scalar and vector reductions. To evaluate RICH, we consider applications with reduction operations on scalar variables as well as parallel codes containing more complex reduction variables composed of arrays. Benchmarks are selected among HPC applications and kernels to cover a wide range of algorithms used in scientific codes. We extend the programming model annotations in the benchmarks to mark the reduction variables as explained in Section 3.2.

Table 1: Benchmark details.

	Benchmark	Short Name	Input Parameters	Reduction task workload size	Reduction data/op type	Reduction instr. ratio	Time spent by redu. instr.
Scalar	Dot Product	DotP	256K elem. 100 iterations	in: 2MB; out: 8B	FP ADD	8.96%	7.95%
	KnightsTour	KT	5×5 chessboard	in: 304K elem.; out: 4B	INT ADD	0.74%	1.88%
	NBinaryWords	NB	word length: 24	in: 2^{24} elem.; out: 4B	INT ADD	14.09%	9.50%
	NQueens	NQ	12 queens on 12×12 chessboard	in: 12! elem.; out: 4B	INT ADD	0.45%	0.30%
	PowerSet	PS	set size: 24 elements	in: 2^{24} elem.; out: 4B	INT ADD	14.37%	11.05%
	Vector Reduction	VectR	256K elem. 50 iterations	in: 2MB, out 8B	FP ADD	22.91%	43.24%
Vector	2D Convolution	2DC	image 1024×1024 pixels, stencil 16×16	in: 16MB; out: 1MB	FP ADD	9.87%	28.09%
	Conjugate Gradient [28]	CG	matrix qa8fm [12], 16 blocks, 97 iterations	in: 529.5KB; out: 516KB	FP ADD	5.82%	1.82%
	Dense Matrx Matrx Multipl.	DGEMM	matrix 1024×1024 elem., block 64×64 elem.	in: 16MB; out: 8MB	FP ADD	1.28%	0.10%
	Dense Matrx Vect. Multipl.	DGEMV	matrix 2048×2048 elem., block 128×128	in: 32MB; out: 16KB	FP ADD	14.21%	16.48%
	2D Expl. Hydro. Frag. [34]	EHF	array 16×64K elem.	in: 9MB; out: 6MB	FP ADD	0.89%	11.14%
	Stencil Histogram	Hist	input: 4MB, 512K bins, 27-point stencil	in: 4MB; out: 2MB	INT ADD	1.31%	11.77%
	LULESH [29]	LULESH	cube 20^3, 11 regions, 10 iterations	in: 1500KB; out: 187.5KB	FP ADD	0.35%	3.00%
	Molecular Dynamics	MD	2k atoms, periodic space, stretch phase change	in: 326KB; out: 163KB	FP ADD	2.34%	0.91%
	N-body Simulation	NBody	4096 bodies, 10 iterations	in: 24MB out: 96KB	FP ADD	5.88%	12.32%
	1D Particle in Cell [34]	PIC	array 128K elem, 8K histogram, 1000 iter.	in: 6.5MB; out: 64KB	FP ADD	17.87%	9.99%
	Sparse Matrix Vect. Multipl.	SpMV	matrix bcsstk32 [12]	in: 12.9MB; out: 357KB	FP ADD	10.63%	15.88%

Table 2: Parameters of the simulated system.

CPU	16 OoO superscalar cores, 128-entry ROB, 2.4GHz, issue width 4
Caches	64B line, non-inclusive, write-back, write-allocate, 16-entry MSHR
L1	private, 32 KB, 8-way set-associative, 4-cycle latency, split I/D
L2	private, 256 KB, 8-way set-associative, 12-cycle latency
L3	shared, 32 MB, 16-way set-associative, 36-cycle latency
RM	16-entry RMIQ, single pipelined FU and ALU
RVT	32 entries per core
Memory	64 cycles + 100ns latency, 85GB/s bandwidth

Table 3: RICH design space exploration.

RM	RMEX: **1**, 2, 4 FUs
	FU design: **pipelined**, non-pipelined
	RMIQ entries: 1, 2, 4, 8, **16**, 32, 64
RVT	1, 2, 4, 8, 16, **32**, 64, 128, 256, 512, 1024, 2048 entries

Table 1 shows the list of all benchmarks used for evaluation including a summary of relevant parameters and properties. We split benchmarks into two groups mentioned in the previous paragraph. Even though CG application performs reduction on both vector and scalar data, we classify it as an application with vector-reductions for simplicity. The third column displays the input parameters used to run each benchmark. The fourth column contains the reduction variable sizes (denoted *out*) and the size of the input structures (denoted *in*). The fifth column shows the reduction variable's data type and the operator for accumulating the values into the reduction variable. Finally, the last two columns show the ratio of executed reduction instructions compared to the overall number of executed instructions and the percentage of overall execution time spent by reduction instructions, respectively.

4.2 Simulation Setup

We use TaskSim, a trace-driven cycle-accurate architecture simulator [39, 40]. TaskSim simulates in detail the execution of parallel applications with OpenMP pragma primitives [36] on parallel multicore environments. The simulated system mimics an Intel Xeon E7 based processor and consists of 16 cores connected to main memory. The cores follow a simple model of a superscalar out-of-order processor with a detailed three-level cache hierarchy. Each core has two private cache levels, L1 and L2, while the L3 is shared. All relevant parameters of the simulated system are shown in Table 2.

Power consumption is evaluated using the McPAT model [33] with a transistor technology of 22 nm, a voltage of 1.2V, and the default clock gating scheme. We incorporate the changes suggested by Xi *et al.* [47] to improve the accuracy of the models. The hardware structures RVT, RMIQ and RMSQ are modeled using CACTI 7 [7]. We add the appropriate counters in TaskSim to measure the extra power introduced by the RM.

5 RICH DESIGN DECISIONS

5.1 Design Space Exploration

There are three design parameters that influence the performance of the proposed Reduction Module: (i) The Reduction Module Instruction Queue size; (ii) Number of the functional units in the RMEX; and (iii) Design of the functional units in the RMEX. In this section, we evaluate the impact of the aforementioned parameters on the processor's performance. In addition, we explore different latencies of arithmetic operations in order to evaluate the performance of the Reduction Module with all supported arithmetic and logic operations on both fixed and floating-point numbers. Operations are modeled to have the same latency as the corresponding instructions in current Intel processors. The list of parameters and explored values is presented in Table 3. For the evaluation we use the benchmarks described in Section 4.1. The final purpose of this

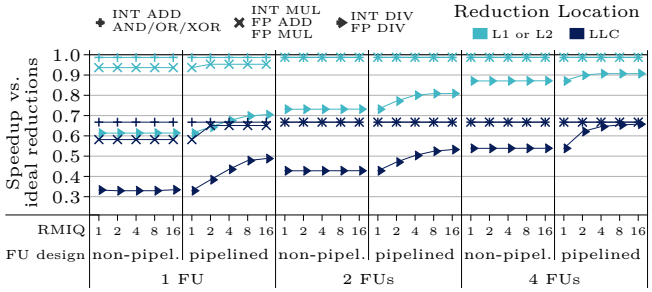

Figure 5: RICH speedup vs. ideal reductions for different configurations of functional units and RMIQ in the RM, depending on operation type and reduction location.

analysis is to determine the optimal parameters of the RM in the context of the simulated processor and evaluated benchmarks.

Figure 5 shows the speedup of RICH versus the ideal implementation of reductions, where each reduction instruction takes 1 cycle and does not interact with the cache subsystem. The speedup of 1 represents the upper theoretical limit for the achievable performance. On x-axis we show different configurations for some of the RM's components. We explore the effects of different counts of functional units and their design (non-pipelined vs. pipelined). The number of RMIQ entries per functional unit is denoted with *RMIQ*.

Each data point shows the mean speedup among all benchmarks for a specific group of reduction operations and the cache level at which the reduction is performed. Operations of similar latencies and behaviors are grouped together. Each group is designated with one of three symbols shown in the left part of the legend. Data corresponding to configurations $RICH_{L1}$ and $RICH_{L2}$ are plotted together as these configurations exhibit similar trends and sensitivity to the RM's parameters. Points belonging to $RICH_{L1}$ and $RICH_{L2}$ are painted in light blue color, while points associated with $RICH_{LLC}$ are presented in dark blue, as shown in the right part of the legend.

The results show that reductions using addition or multiplication exhibit little sensitivity to the RM's configurations due to relatively low operation latency. Division, however, benefits from having more functional units and a larger RMIQ. Using two FUs in the RMEX improves performance of $RICH_{L1}$ and $RICH_{L2}$ by 2.7% on average compared to the single FU-design, but comes with 94.2% higher area overhead. Pipelined functional units benefit from more RMIQ entries as they are able to execute multiple independent operations simultaneously, contrary to the non-pipelined designs. Considering these results, we use a single pipelined ALU and a single pipelined FPU. We decide to have an RMIQ with 16 entries to get best possible division performance. This configuration will be used for the further evaluation of RICH presented in the remaining of this document.

Reduction Variable Table (RVT) is used by the core to recognize the instructions involved in the reduction and is described in detail in Section 3.1. To evaluate the impact of the RVT design on the processor's performance, we model the RVT using CACTI 7 based on the chip frequency of 2.4 GHz. The model shows that small RVT designs with up to 256 entries can be accessed within 1 cycle,

Table 4: Hardware cost of implementing RICH in 22nm.

	RVT	RMIQ	$RMEX_{ALU}$	$RMEX_{FPU}$	RMSQ
Area [mm^2]	0.0002	0.013	0.038	0.223	0.003
Storage [KB]	0.055	3.22	-	-	2.09

Baseline processor's area		192.48	mm^2
Reduction Module area	(IQ + EX + SQ)	0.277	mm^2
Total area overhead	(17 RMs + 16 RVTs)	4.71	mm^2
		2.45	%

while medium (up to 1024 entries) and larger designs require 2 and 3 cycles respectively. RVT configurations with latencies of 2 and 3 cycles degrade the overall performance by negligible 0.8% and 1.8% on average, respectively, compared to the RVT with 1-cycle latency. Applications having more in-flight reduction variables that cannot fit in the RVT are still executed correctly. In that case, reduction operations on variables that do not fit into the RVT are not accelerated by the RM. To optimally run all benchmarks from Table 1, a 4-entry RVT is needed. We select a design with a 32-entry RVT as it covers potentially more complex codes while keeping 1-cycle access latency.

5.2 Hardware Cost of Implementing RICH

In this section we discuss the area and storage required to implement RICH. As explained in Section 3.1, the Reduction Module (RM) consists of two queues, RMIQ and RMSQ, and an Execution Unit (RMEX) that contains two functional units, i.e. one ALU and one FPU. Private caches L1 and L2 share a single Reduction Module and the LLC has its own RM. Additionally, the design has a single Reduction Variable Table (RVT) per core. Thus, the simulated 16-core processor contains 17 Reduction Modules and 16 RVTs.

Table 4 shows the sizes of particular RM components as well as overall area of a processor occupied by the added hardware. According to the McPAT model, the FPU used in the RM is 60% smaller than the FPU in the core due to the removal of support for SIMD instructions. The modeled ALU supports 32bit and 64bit arithmetic and logic operations on integers.

The total area consumed by the RMs is 4.71 mm^2 or 2.4% of the whole die area of the baseline processor. Alternatively, we consider a processor design that uses a larger LLC. Our simulations show that a processor with a 40 MB, 16-way set-associative LLC obtains on average 1.5% better performance than the reference processor. RICH performs 51.1% better than the baseline processor while requiring 30% less additional chip area than a 40 MB LLC. Thus, we conclude that RICH utilizes additional hardware more efficiently than just extending already existing hardware components like the LLC.

6 EVALUATION

6.1 Evaluating RICH with Vector-Reductions

The analysis in Section 2 shows that, in case of larger arrays, reductions implemented with atomics achieve better performance than software privatization. Taking this into consideration, we choose atomics as the baseline for evaluating RICH on benchmarks that perform reductions on vector variables. All reported results in the following sections correspond to the overall performance including both reduction and non-reduction tasks.

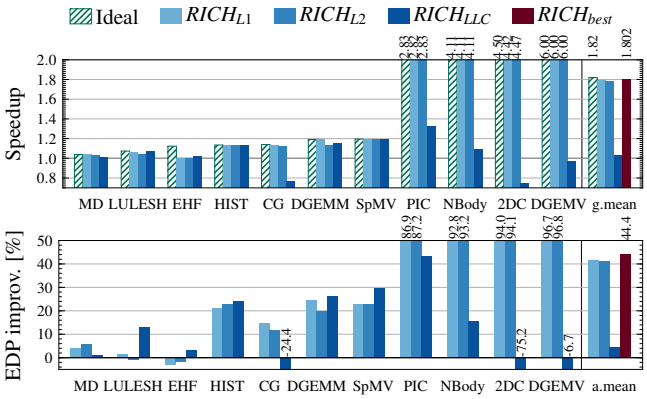

Figure 6: Speedup and Energy-Delay Product improvement of RICH over the baseline with atomics for benchmarks with reductions on arrays.

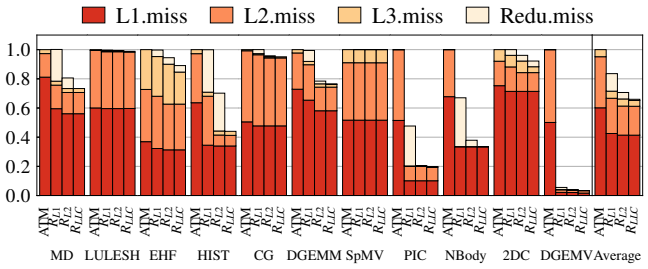

Figure 7: Breakdown of misses across all cache levels. `Redu.miss` denotes misses generated by the Reduction Module in the cache level where the reduction is performed. Configurations *Atomics*, $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$ are denoted as ATM, R_{L1}, R_{L2} and R_{LLC}, respectively.

The upper part of Figure 6 shows performance speedups of the three RICH configurations normalized to the reduction approach based on atomic operations. We also show performance of ideal reductions, where each reduction operation takes 1 cycle and does not issue requests to the cache hierarchy. The ideal configuration indicates the maximal achievable speedup per benchmark.

On average, $RICH_{L1}$, $RICH_{L2}$ and $RICH_{LLC}$ perform 1.79×, 1.76× and 1.03× faster than the baseline, respectively. In general, RICH outperforms the implementation with atomics due combination of several factors. (i) RICH performs the load-modify-store sequence as one instruction, and therefore reduces the number of requests to the cache hierarchy and does not use ALUs and FPUs in the core. (ii) RICH does not suffer from coherence effects caused by conflicts between different threads, contrary to the reductions with atomics. Coherence effects manifest themselves in increased miss ratio to reduction variable due to invalidations by other threads and retrying the update operation or waiting for a lock release, depending on the implementation of atomics in the target architecture. (iii) When the reduction variable is updated at higher cache levels, it is not present in the lower cache levels, reducing the pollution of these caches. This effect results in better cache performance for input data. Additionally, due to larger sizes of higher cache levels, accesses to the reduction variable result in less misses, which is explained in the following section. These three factors contribute to, on average, lower execution time of RICH compared to atomics.

The highest performance gains are observed in PIC, NBody 2DC and DGEMV, where $RICH_{L1}$ performs from 2.8× to 6.0× faster than atomics-based approach. The main contributor for faster execution in case of PIC, NBody and DGEMV is the reduced number of misses, as shown in Figure 7. On the other hand, for 2DC we observe a small reduction in cache misses. Even though collisions still occur, RICH reduces amount of cycles spent on waiting due to lock contention.

The lower part of Figure 6 shows the improvements in energy-delay product (EDP) of RICH compared to the baseline with atomics. On average, the best RICH configuration per benchmark improves EDP by 44.4% compared to the baseline. The highest EDP improvements are observed for the benchmarks where RICH achieves highest speedups, ie. PIC, NBody, 2DC and DGEMV with 87.2% to 96.8% improvement. EDP is mainly improved due to lower execution time and reduced energy consumption by the caches due to reduced amount of misses in RICH configurations, which is demonstrated in Section 6.2.

$RICH_{LLC}$ consumes less power than $RICH_{L1}$ and $RICH_{L2}$ since it uses just one RM in the LLC. This effect is clearly observed for EHF, Hist and SpMV. Even though these benchmarks achieve similar performance across all RICH configurations, there are notable differences in the EDP among three RICH configurations. The additional reason for such behavior is the reduced number of misses in $RICH_{L2}$ and $RICH_{LLC}$ configurations, as we describe in Section 6.2.

We define $\boldsymbol{RICH_{best}}$ as the optimal RICH configuration per benchmark. In benchmarks reducing on vector variables, $RICH_{best}$ achieves performance speedup of 1.8× and 44.4% better EDP than atomics. Our proposal allows the programmer to specify the optimal reduction location via pragma constructs supported by the programming model, as we describe in Section 3. In this context, $RICH_{best}$ represents the performance improvement that can be obtained by choosing the best location to carry out the reductions.

6.2 Impact of RICH on Cache Performance

Figure 7 shows the breakdown of misses for all three cache levels regarding benchmarks that perform reductions on vectors. Misses are normalized to the total misses occurring when reductions rely on atomic operations (configuration *ATM*). Label `Redu.miss` corresponds to the misses triggered by accesses to the reduction variable generated by the Reduction Module. These misses occur in the cache level where the reduction is performed.

In eight benchmarks there is a negligible difference in total misses between *ATM* and $RICH_{L1}$. In these cases, the reduction variable is accessed in a more structured manner which does not cause data invalidations invoked by the coherence protocol. Nonetheless, $RICH_{L1}$ still achieves speedup over atomics due to time penalties when using atomic instructions, in addition to the fact that RICH internally compacts the load-modify-store instructions into one instruction. For other benchmarks, we can observe the effects of coherence invalidations that manifest themselves as increased total number of misses in *ATM* compared to $RICH_{L1}$. This is most notable in benchmarks where RICH achieves highest performance speedups, ie. DGEMV, NBody and PIC.

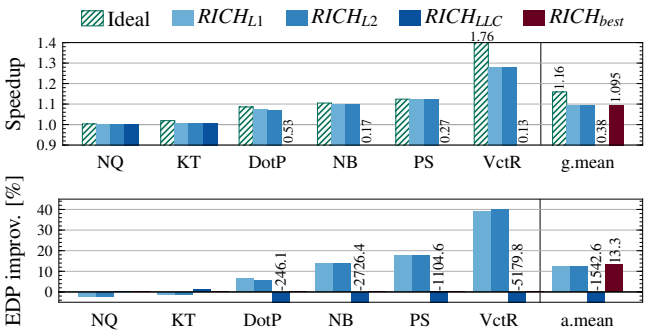

Figure 8: Speedup and Energy-Delay Product of RICH compared to the baseline with software privatization for benchmarks that perform reductions on scalars.

Another interesting effect to analyze is the significant average reduction of total number of misses when performing reductions in higher cache levels, i.e. the L2 and the LLC. The cause for this behavior is the reduced pollution of the L1 cache by the reduction variable and higher hit ratio to the reduction variable in L2/LLC due to larger size of those caches. This effect is observable in almost all benchmarks and is most prominent in MD, DGEMM, Hist, NBody and PIC. The reduction in misses is not translated into performance improvements of $RICH_{L2}$ over $RICH_{L1}$ because the added miss penalties are hidden by an out-of-order core. However, as having less misses results in reduced cache traffic, the energy consumed by the memory hierarchy is reduced, which is demonstrated through EDP improvements in Section 6.1.

6.3 Evaluating RICH with Scalar-Reductions

As shown in Section 2, privatization is the best performing technique for handling reductions in applications with reductions on scalar variables. Therefore, we select software privatization as the baseline for parallel codes that perform reductions on scalars.

The top part of Figure 8 shows the performance speedup of three RICH configurations normalized to software privatization. On average, $RICH_{L1}$ and $RICH_{L2}$ perform 1.095× faster than the baseline. RICH achieves the highest performance benefits for applications that have highest ratio of reduction instructions with respect to the overall number of instructions, such as DotP, NB, PS and VctR. The performance benefits come from the reduced number of instructions executed in the core. Specifically, since reduction operations are offloaded to the Reduction Module (RM), the core can execute instructions in advance while the RM computes the reduction in the cache. NQ and KT exhibit marginal improvements since the amount of instructions involved in reduction operations of these benchmarks represents a small percentage of the whole execution.

We also show the performance of an idealistic implementation of reductions, where each reduction operation is performed instantaneously and does not issue requests to the cache hierarchy, as described in Section 6.1. Results show that RICH achieves close-to-ideal performance in all benchmarks on scalars except VctR, a benchmark that calculates a sum of double-precision floating point values on a scalar variable of the same type. Since we model this operation to take 3 cycles, the serialization of reductions in the RM

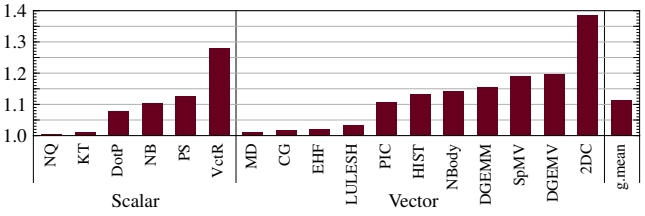

Figure 9: Speedup of $RICH_{best}$ compared to COUP [50].

combined with the high frequency of reduction instructions in this benchmark limits the performance achieved by RICH.

In $RICH_{L2}$ configuration, fetching data from the L2 cache to the RM takes more cycles than the equivalent operation in $RICH_{L1}$ configuration due to the higher access latency of the L2 cache. Nonetheless, we observe the same performance for these configurations. The explanation for this behavior is the fact that the execution of reduction instructions in $RICH_{L1}$ and $RICH_{L2}$ configurations is overlapped with other instructions executed at the CPU level in a way that the reduction latencies are hidden.

$RICH_{LLC}$ suffers from performance slowdowns compared to the baseline. With reductions on scalar variables at the LLC, all reduction instructions are serialized as they depend on each other. This explains the performance slowdown suffered by DotP, NB, PS and VctR. Contrarily, this effect is not visible in NQ and KT. Due to the low ratio of reduction instructions in these benchmarks, serialized instructions from one iteration in the LLC's RM have time to finish before the arrival of instructions from the next iteration. Moreover, the benefits of offloading instructions to the RM outweigh the small performance degradation due to serialization in the LLC.

The bottom part of Figure 8 shows the improvements in energy-delay product (EDP) of RICH compared to the baseline with software privatization. On average, $RICH_{best}$ improves EDP by 13.3% compared to the baseline. The main factor contributing to EDP improvements is faster execution time, particularly for the four benchmarks where RICH achieves notable speedups. In the case of NQ and KT, $RICH_{LLC}$ achieves the best EDP due to less power overhead of having just one RM in the LLC compared to having one RM per core in $RICH_{L1}$ and $RICH_{L2}$ configurations.

6.4 Comparison with Other Proposals

In this section, we compare $RICH_{best}$ with the state-of-the-art technique for reductions in hardware, COUP [50]. $RICH_{best}$ is defined as the best RICH configuration per benchmark in terms of performance. COUP implements privatization of the reduction variable in the private caches by modifying the cache coherence protocol. This design allows multiple cores to acquire a line with update-only permission. The partial results are accumulated in private caches using in-core functional units, while the final result is calculated on demand in the LLC or memory controller, which are equipped with dedicated functional units. To simulate COUP, we mimic its functionality in the context of our simulation infrastructure. We assume coherence operations performed by COUP to have zero cost. The handling of *update-only* lines is implemented in detail.

Figure 9 shows the speedup of $RICH_{best}$ compared to COUP. $RICH_{best}$ achieves 1.11× better performance on average and up

to 1.38× improvement in case of 2DC. Significant improvements are also obtained for Hist, NBody, DGEMM, SpMV and DGEMV. RICH outperforms COUP due to reducing the traffic between the core and the L1 cache as the reduction variable is updated directly in the cache. Moreover, the ability to execute reductions at higher cache levels benefits benchmarks like LULESH, 2DC and VctR.

According to the McPAT and CACTI models, RICH requires 2.45% more area than the baseline processor and introduces 3.8 % of power overhead. However, the performance improvement of 1.11× over COUP compensates for the increased power consumption of the RICH design. Consequently, RICH achieves better energy consumption than COUP. Another important improvement of RICH over COUP is the support for external pre-compiled libraries. Many scientific applications use mathematical libraries that implement algorithms with reductions, e.g. matrix multiplications. COUP requires modifying the ISA to mark the loads and stores belonging to the reduction operation, which requires access to the complete source code to be compiled. RICH uses information about the reduction variable provided by the runtime system and does not require ISA modifications, thus supporting linking against pre-compiled algorithmic libraries.

7 RELATED WORK

Software techniques that support reduction operations fall into the categories of direct access or techniques that reorder iterations. In particular, the use of atomic updates is a technique that implements direct accesses to either scalar or array reduction variables. LocalWrite [48] reorders iterations to avoid data races. PAE [20] and SelectPriv [21] apply privatization selectively to mitigate effects of concurrent updates while minimizing overheads due to privatization. LocalWrite and SelectPriv require the knowledge of the iteration space, making them applicable only to algorithms with a static iteration space [22]. RICH does not suffer from these restrictions and is applicable to any iterative construct. PIBOR [9] combines privatization and redirection to achieve linear updates to private copies of reduction variable. OmpSs-RM [10] formalizes support of the software techniques for declarative parallel programming models. Unlike RICH, above-described software techniques can suffer from negative effects of privatization and do not offer a mechanism to select optimal reduction location.

ARMv8 ISA [6] offers vector instructions for reductions as a part of the SIMD extensions. The list of supported operators is limited and depends on the data type. Reductions on scalar variables are easily supported while reductions on vectors require additional effort from the programmer or the compiler, especially for applications with more complex access patterns. Moreover, operations are performed in the core and, thus, do not reduce data movement in the memory hierarchy.

Transactional Memory (TM) [23] offers mechanisms that can be used for implementing reductions [18]. Software TM implementations utilize existing lock and atomic operations and, thus, have similar drawbacks as other previously discussed solutions that use these operations. Hardware-accelerated TM (HTM) handles conflicts with speculative execution and rollback mechanism, which wastes energy in codes with frequently occurring conflicts.

Massively parallel processors (GPUs) offer primitives used by algorithmic proposals for efficient execution of reductions [13, 15]. While these approaches are effective on GPUs owing to efficient synchronization and lock-step execution inside a warp, they are not applicable to general purpose processors, where these mechanisms are not present.

Previously proposed hardware solutions, such as COUP [50], CommTT [49] and PCLR [17], implement on-demand privatization of reduction variable in on-chip private caches. As a consequence of relying on privatization, these designs suffer from the same problems as software based approaches, e.g. cache pollution. PHI [35] is another hardware-based approach for coalescing and buffering of scattered updates in private caches. Similarly to RICH, PHI adds functional units inside the cache controllers. RICH offers programming model extensions to facilitate simple designation of the reduction variable, while none of the mentioned hardware-based prior works offer such a mechanism. Contrary to RICH, these proposals require ISA extensions to manually mark the reduction instructions, which limits their use in real systems in the proposed context. Moreover, RICH gives a programmer options to select the optimal reduction technique for a given scenario. Previous designs use fixed configurations, which might not be the optimal solution for each application. Finally, unlike RICH, COUP requires changes to the cache coherence protocol. Complexity of coherence protocol validation increases dramatically with higher number of cores. Therefore, RICH is better suited for modern and future many-core processors.

Scatter-Add in data parallel architectures [3] targets reductions in SIMD/vector/stream memory systems [11]. Many designs implement atomic operations beyond the private caches, such as the LLC [5], memory controller [16, 26, 30, 32, 42, 51] and network switches [19]. The operations are restricted to integer additions and logic operations. This mechanism allows an efficient implementation of synchronization primitives, such as barriers and locks. However, the performance is limited for more complex reduction operations encountered in codes from the HPC domain. Solutions [43, 44] that exploit the properties of DRAM technologies to offload reduction-like operations to the main memory are limited only to simple operations. On the contrary, RICH supports all commonly used reduction operations.

8 CONCLUSIONS

In this work we present RICH, a proposal to accelerate the execution of reductions on modern processors. RICH improves the performance of vector-reductions while keeping well-performing support for reductions on scalars. RICH enables the programmer to select the optimal cache level where reductions take place. It relies on hardware, runtime system and OpenMP-compatible programming model extensions.

Extensive evaluation with reductions on vector variables show that RICH outperforms the atomics-based software technique in terms of execution speed on average by 1.8× and up to 6.0×. The energy-delay product is improved up to 96.8% (44.4% on average). Moreover, the total number of misses in the cache hierarchy is reduced by up to 96.6% (34.0% on average). RICH implementation requires only 2.4% additional silicon area and introduces a 3.8% power overhead.

The results show that executing reductions in the private caches performs significantly better compared to the case where their execution is centralized in the last-level cache. In addition, the performance is similar for both configurations that execute reductions in the private caches. A fabricated chip could implement only one of the two well-performing RICH configurations.

RICH outperforms COUP, a state-of-the art hardware-based technique for reductions, by up to 1.38× (1.11× on average). Furthermore, thanks to its runtime-hardware interaction, RICH does not modify the ISA. Thus, it is compatible with applications that use routines with reductions present in pre-compiled mathematical and algorithmic libraries.

ACKNOWLEDGMENTS

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), and by Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

V. Dimić has been partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under Ajuts per a la contractació de personal investigador novell fellowship number 2017 FI_B 00855. M. Moretó has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowship number RYC-2016-21104. M. Casas has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2017-23269. This manuscript has been co-authored by National Technology & Engineering Solutions of Sandia, LLC. under Contract No. DE-NA0003525 with the U.S. Department of Energy/National Nuclear Security Administration.

We appreciate the suggestions by anonymous reviewers which helped us improve the quality of this work. We thank to Vicenç Beltran and Sergi Mateo for sharing their valuable experience with reductions in the context of programming models, to Lluc Alvarez, Luc Jaulmes and Francesc Martinez for numerous technical discussions, and to Dimitrios Chasapis for the help on improving the writing of this paper.

REFERENCES

[1] Advanced Micro Devices. 2018. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions.* Technical Report 24594. Advanced Micro Devices.

[2] Vikas Aggarwal, Yogish Sabharwal, Rahul Garg, and Philip Heidelberger. 2009. HPCC RandomAccess benchmark for next generation supercomputers. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, USA, 1–11. https://doi.org/10.1109/IPDPS.2009.5161019

[3] Jung Ho Ahn, Mattan Erez, and William J. Dally. 2005. Scatter-Add in Data Parallel Architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05).* IEEE, USA, 132–142. https://doi.org/10.1109/HPCA.2005.30

[4] ARM 2013. *ARM® Architecture Reference Manual. ARMv8, for the ARMv8-A architecture profile.* ARM.

[5] ARM 2016. *ARM® Cortex®-A75 Core. Technical Reference Manual.* ARM.

[6] ARM 2018. *ARM® Architecture Reference Manual Suplement. The Scalable Vector Extension (SVE), for ARMv8-A.* ARM.

[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (Jun 2017), 25 pages. https://doi.org/10.1145/3085572

[8] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. 1994. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.* Springer-Verlag, Berlin/Heidelberg, 10–1.

[9] Jan Ciesko, Sergi Mateo, Xavier Teruel, Vicenc Beltran, Xavier Martorell, and Jesus Labarta. 2015. Boosting irregular array Reductions through In-lined Block-ordering on fast processors. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, MA, USA). IEEE, USA, 1–6. https://doi.org/10.1109/HPEC.2015.7322443

[10] Jan Ciesko, Sergi Mateo, Xavier Teruel, Xavier Martorell, Eduard Ayguadé, and Jesus Labarta. 2016. Supporting Adaptive Privatization Techniques for Irregular Array Reductions in Task-Parallel Programming Models. In *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP* (Nara, Japan) *(IWOMP 2016)*, Vol. 9903. Springer, Cham, 336–349. https://doi.org/10.1007/978-3-319-45550-1_24

[11] William J. Dally, François Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (Phoenix, AZ, USA) *(SC '03).* ACM, New York, NY, USA, 35. https://doi.org/10.1145/1048935.1050187

[12] Timothy Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

[13] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019).* IEEE Press, Piscataway, NJ, USA, 73–84.

[14] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. 1974. Design of Ion-implanted MOSFETs with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (October 1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

[15] Ian J. Egielski, Jesse Huang, and Eddy Z. Zhang. 2014. Massive Atomics for Massive Parallelism on GPUs. In *Proceedings of the 2014 International Symposium on Memory Management* (Edinburgh, United Kingdom) *(ISMM '14).* ACM, New York, NY, USA, 93–103. https://doi.org/10.1145/2602988.2602993

[16] Zhen Fang, Lixin Zhang, John B. Carter, Sally A. McKee, Ali Ibrahim, Michael A. Parker, and Xiaowei Jiang. 2012. Active memory controller. *The Journal of Supercomputing* 62, 1 (01 October 2012), 510–549. https://doi.org/10.1007/s11227-011-0735-9

[17] María Jesús Garzarán, Milos Prvulovic, Ye Zhang, Josep Torrellas, Alin Jula, Hao Yu, and Lawrence Rauchwerger. 2001. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01).* IEEE Computer Society, USA, 243. https://doi.org/10.1109/PACT.2001.953304

[18] Miguel A. Gonzalez-Mesa, Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. 2013. Exploring Irregular Reduction Support in Transactional Memory. In *Algorithms and Architectures for Parallel Processing*, Joanna Kołodziej, Beniamino Di Martino, Domenico Talia, and Kaiqi Xiong (Eds.). Springer International Publishing, Cham, 257–266. https://doi.org/10.1007/978-3-319-03859-9_22

[19] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1983. The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.* C-32, 2 (February 1983), 175–189. https://doi.org/10.1109/TC.1983.1676201

[20] Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata. 2001. Improving Parallel Irregular Reductions Using Partial Array Expansion. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, CO) *(SC '01).* ACM, New York, NY, USA, 56–56. https://doi.org/10.1145/582034.582072

[21] Hwansoo Han and Chau-Wen Tseng. 1999. Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC '98).* Springer-Verlag, London, UK, 181–196. https://doi.org/10.1007/3-540-48319-5_12

[22] Hwansoo Han and Chau-Wen Tseng. 2001. A Comparison of Parallelization Techniques for Irregular Reductions. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS '01).* IEEE, USA, 27.

[23] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, CA, USA) *(ISCA '93).* ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[24] Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (July 1999), 355–372. https://doi.org/10.1017/S0956796899003500

[25] IBM Corporation. 2017. Power ISA Version 3.0 B.

[26] IBM Corporation. 2018. Power9 Processor User's Manual. version 2.0.

[27] Intel Corporation 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation.

[28] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. ACM, New York, NY, USA, Article 53, 12 pages. https://doi.org/10.1145/2807591.2807599

[29] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory. 1–9 pages.

[30] Richard E. Kessler and Jim L. Schwarzmeier. 1993. Cray T3D: a new dimension for Cray Research. In *Digest of Papers. Compcon Spring* (San Francisco, CA, USA). IEEE, USA, 176–182. https://doi.org/10.1109/CMPCON.1993.289660

[31] Dimitri Komatitsch and Jeroen Tromp. 1999. Introduction to the spectral-element method for 3-D seismic wave propagation. *Geophysical Journal International* 139, 3 (1999), 806–822.

[32] James Laudon and Daniel Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 241–251. https://doi.org/10.1145/384286.264206

[33] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[34] Francis H. McMahon. 1986. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range.* Technical Report UCRL-53745. Lawrence Livermore National Laboratory, Livermore, CA.

[35] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 1009–1022. https://doi.org/10.1145/3352460.3358254

[36] OpenMP Architecture Review Board. 2016. OpenMP Technical Report 4 Version 5.0 Preview 1.

[37] William M. Pottenger. 1998. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-level Parallelism. In *Proceedings of the 12th International Conference on Supercomputing* (Melbourne, Australia) *(ICS '98)*. ACM, New York, NY, USA, 188–195. https://doi.org/10.1145/277830.277870

[38] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. 2000. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design* (Rapallo, Italy) *(ISLPED '00)*. ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/344166.344526

[39] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. 2012. On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels. *ACM Trans. Archit. Code Optim.* 8, 4, Article 36 (January 2012), 20 pages. https://doi.org/10.1145/2086696.2086715

[40] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. 2011. Trace-driven simulation of multithreaded applications. In *Performance Analysis of Systems and Software, 2011 IEEE International Symposium on* (Austin, TX, USA) *(ISPASS'11)*. IEEE, USA, 87–96. https://doi.org/10.1109/ISPASS.2011.5762718

[41] RISC-V Foundation 2017. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.* RISC-V Foundation. Editors Andrew Waterman and Krste Asanović.

[42] Steven L. Scott. 1996. Synchronization and Communication in the T3E Multiprocessor. *SIGPLAN Not.* 31, 9 (September 1996), 26–36. https://doi.org/10.1145/248209.237144

[43] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/2540708.2540725

[44] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. ACM, New York, NY, USA, 273–287. https://doi.org/10.1145/3123939.3124544

[45] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable ProgrammerâĂŹs Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[46] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.* 29, 12 (December 1994), 31–37. https://doi.org/10.1145/193209.193217

[47] Sam Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *International Symposium on High Performance Computer Architecture (HPCA)* (Burlingame, CA, USA). IEEE, USA, 577–589. https://doi.org/10.1109/HPCA.2015.7056064

[48] Hao Yu and Lawrence Rauchwerger. 2000. Adaptive Reduction Parallelization Techniques. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2591635.2667180

[49] Guowei Zhang, Virginia Chiu, and Daniel Sanchez. 2016. Exploiting semantic commutativity in hardware speculation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, USA, 1–12. https://doi.org/10.1109/MICRO.2016.7783737

[50] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-coherent Systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/2830772.2830774

[51] Lixin Zhang, Zhen Fang, and John B. Carter. 2004. Highly efficient synchronization based on active memory operations. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, USA, 58–67. https://doi.org/10.1109/IPDPS.2004.1302981