ATM: Approximate Task Memoization in the Runtime System

Iulian Brumar, Marc Casas, Miquel Moreto, Mateo Valero
*Barcelona Supercomputing Center (BSC), Barcelona, Spain
Email: {name.surname@bsc.es}

Gurindar S. Sohi
†University of Wisconsin-Madison, USA
Email: sohi@cs.wisc.edu

***Abstract*—Redundant computations appear during the execution of real programs. Multiple factors contribute to these unnecessary computations, such as repetitive inputs and patterns, calling functions with the same parameters or bad programming habits. Compilers minimize non useful code with static analysis. However, redundant execution might be dynamic and there are no current approaches to reduce these inefficiencies. Additionally, many algorithms can be computed with different levels of accuracy. Approximate computing exploits this fact to reduce execution time at the cost of slightly less accurate results. In this case, expert developers determine the desired tradeoff between performance and accuracy for each application.**

In this paper, we present Approximate Task Memoization (ATM), a novel approach in the runtime system that transparently exploits both dynamic redundancy and approximation at the task granularity of a parallel application. Memoization of previous task executions allows predicting the results of future tasks without having to execute them and without losing accuracy. To further increase performance improvements, the runtime system can memoize similar tasks, which leads to task approximate computing. By defining how to measure task similarity and correctness, we present an adaptive algorithm in the runtime system that automatically decides if task approximation is beneficial or not. When evaluated on a real 8-core processor with applications from different domains (financial analysis, stencil-computation, machine-learning and linear-algebra), ATM achieves a 1.4x average speedup when only applying memoization techniques. When adding task approximation, ATM achieves a 2.5x average speedup with an average 0.7% accuracy loss (maximum of 3.2%).

I. Introduction

During the execution of an application, redundant computations appear at several levels of granularity, from fine-grained instructions to coarse-grained library or program calls. Value Prediction (VP) [13] is a technique that detects patterns in the results produced by a static instruction and bypasses this instruction when it is very likely to produce a redundant output. VP has been very effective in eliminating redundancy at the instruction-level, especially in the case of memory operations. If the result was actually not redundant the technique incurs recovery overheads. Memoization takes into account the inputs of the instructions and eliminates the need of recovery [21], but releases lesser amounts of parallelism as it is done in a later stage of the pipeline.

Approximate computing brings improvements in terms of performance, energy consumption, fault tolerance and efficient parallel execution by allowing some accuracy losses [11], [15], [19]. In the last decade for instance, approximate versions of VP and memoization at instruction level have been suggested, returning promising performance results [3], [15].

With multi-core processors, the performance of applications can be increased by exploiting thread level parallelism. To ease the complexity of programming multicore processors, a runtime system layer between the program and the architecture has become common [7] [22]. A runtime system decouples the concept of thread from the task which is the computational working unit. Task-based dataflow programming models exploit irregular parallelism by requiring annotations about task input and output data [1], [10].

In this paper, we propose Approximate Task Memoization (ATM) in order to further increase the performance of programs by allowing the runtime system to eliminate redundant computations. ATM leverages task data information available in the runtime system to identify tasks that can be memoized. We find out that tasks with coarse inputs and outputs (up to 4MB) produce redundant results because of repetitive patterns in the application input sets and because the algorithms converge faster on some pieces of their data structures. Our ATM implementation deployed on a real multi-core processor automatically detects these redundant computations, bringing substantial performance improvements as a result.

To further boost its potential, we allow ATM to memoize similar tasks by studying the relationship between the task inputs and outputs. Task approximation delivers more chances of increased performance while reducing the overhead in the runtime system. Moreover, we show how the runtime system can automatically control the degree of task approximation, by bounding correctness loss and achieving high performance.

The main contributions of the paper are:

- **Static ATM**. We implement task memoization at coarse granularities (up to 4MB of task input data) in the runtime system. A hashing mechanism distinguishes tasks and allows checking for redundancy only against tasks likely to produce similar results.
- **Dynamic ATM**. Some programs have low task reuse when considering the whole task inputs. We implement task approximation at the runtime system level that sup-

 DOI: 10.1109/IPDPS.2017.49

ports multiple degrees of similarity. Using simplified task input selection mechanisms, ATM increases reuse and diminishes hashing overheads. We show how the runtime system automatically decides the percentage of task inputs in a training phase, where we just emulate memoization and check whether the task outputs are still within acceptable ranges of correctness by using a sensible distance metric in high output dimensionalities.
- **An extensive evaluation** on a real 8-cores system. We show that Static ATM achieves a $1.4\times$ average speedup and 0% accuracy loss and Dynamic ATM a $2.5\times$ average speedup and accuracy losses below 3.2% (0.7% on average) in the overall execution of the application. A careful lock design to avoid synchronization overheads in the runtime system allows ATM to scale well as we increase the number of cores in the system.

This paper is organized as follows. Section II describes the related work. Section III presents our approaches for task memoization and approximation in the runtime system. The experimental setup and evaluation results are detailed in Sections IV and V respectively. Finally, Section VI draws the main conclusions of this work.

II. Related Work

Redundancy elimination has been investigated in the computer architecture community from the point of view of value locality and prediction [13], [18]. In the compilers community, it has also been studied from the point of view of instruction reuse and memoization [9], [20]. According to previous classifications [21], ATM falls into the category of memoization because it makes use of the task inputs for identifying reuse opportunities rather than trying to detect patterns in the outputs and then predict. The advantage of memoization is that no recovery is needed when the memoized instruction does not have side effects. We are not aware of any approach proposing memoization schemes at the task granularity in the runtime system.

A. Data Reuse and Memoization

Sastry et al. [17] bypass redundant chains of dynamic instructions that are large enough for improving performance via memoization, and evaluate the potential benefits using a model. In contrast, we provide an online implementation on a real multi-core that can memoize all the independent chains of instructions contained in a task.

Earlier proposals link instructions at hardware level to increase the benefits of memoization and reduce their overheads [20], achieving instruction reuses of up to 50%. Other compiler techniques achieve instruction reuse at granularities coarser than a basic block [9], allowing the compiler to identify regions that are good candidates for reuse. This information is communicated to the micro-architecture during execution, which bypasses the execution of those regions that have executed in the history with the same inputs.

B. Approximate Computing

Many algorithms essentially produce approximate solutions for some complex problems [5], while many applications can tolerate reduced accuracy losses [19]. Application experts are using specific techniques to relax the correctness requirements of the applications and get performance and even power gains. Approximate computing avoids code deterioration incurred by those specific techniques by using cross-domain approaches.

1) Tolerating Accuracy Loss: Miguel et al. [15] tolerate some accuracy loss when VP misspredicts, while the approximation mechanism becomes more conservative. This micro-architectural enhancement does not require any offline profiling, but relies on the programmer to identify the loads that can be approximated. Similarly, but at the runtime level, ATM does not need profiling and relies on the programmer to specify the task types appropriate for approximation. This guidance from the programmer is feasible as has been previously acknowledged [11].

Proposals such as loop perforation [19] and HELIX-UP [6] are compiler-based approximation techniques that require nearly no programmer intervention. However, these techniques require a significant amount of profiling time for calibration on representative inputs. In loop perforation there is no runtime metric to check whether the program is still within acceptable accuracy limits, and only relies on profiling. The authors acknowledge that approximate techniques produce errors that can be magnified or diminished at runtime when the approximated results are consumed by other parts of the program. In contrast, HELIX-UP benefits from giving flexibility to a runtime that relaxes the semantics of an automatic loop parallelizer.

2) Accelerators and Low Power Computing: Redundant values do not have to be exactly equal to lead to the same or similar program result. This aspect has been exploited in the accelerators community, where a reduced degradation in accuracy is traded off for performance in some parts of the pipeline [14], [16]. This is common for example in signal processing and machine learning architectures. Fuzzy memoization [3] uses the same output for instructions that have the same most significant bits in the mantissa. However ATM approximates at the coarse granularity of a task.

A different approach consists of optimizing any programmer-selected function with an accelerator that behaves as a neural network with weights assigned with offline profiling [11]. ATM shares with this work the need of a programmer choosing a hot region of code suitable for approximation. However the approach in [11] only approximates regions with statically defined inputs sizes while ATM accepts inputs sizes that vary at execution time.

C. Task-based Dataflow Programming Models

Task-based dataflow programming models such as OpenMP 4.0 [1] conceive the execution of a parallel program

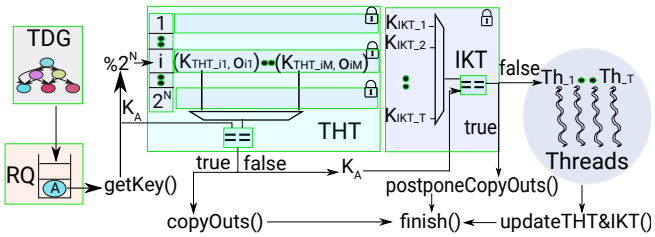


Figure 1. Data structures in the runtime system to support Approximate Task Memoization. TDG stands for Task Dependence Graph, RQ for Ready Queue, THT for Task History Table and IKT for In-flight Key Table. Each bucket 'i' in the THT contains keys K_{THT_ij} and task outputs o_{ij}.

as a set of tasks with dependences among them. Typically, the programmer adds code annotations to split the serial code into tasks that can potentially run in parallel. Each annotation defines a different *task type*. Also, the programmer specifies what data is used by each task (called data inputs) and what data is produced (called data outputs). The runtime system is in charge of managing the execution of the tasks, based on the data dependences, by building a *task dependence graph* (TDG), a directed acyclic graph where nodes represent tasks and edges dependences among them. The runtime system schedules a task on a core when all its dependences are ready and, when the execution of the task finishes, the dependent tasks become ready for execution.

Task-based dataflow programming models are specially well suited for task memoization and approximation. The complete specification of the data inputs and outputs allows the runtime system to identify redundant task executions. Also, the distance between the outputs of a particular task type can be computed to measure precision. Based on this information, in the next section we introduce an adaptive mechanism able to automatically trigger task memoization and approximation without compromising the accuracy of the program.

III. Approximate Task Memoization Framework

A. Task Memoization in the Runtime System

Next, we describe how task memoization can be implemented in the runtime system. To minimize redundant execution of tasks, task memoization should store the history of all task executions including their inputs and outputs. When a new task is ready for execution, we would check in the whole history if the same task type with the same data inputs has been executed. If this was the case, then we would provide the outputs of the current task using the outputs of the previously executed task. However, the overheads of maintaining, accessing and updating this data structure would become prohibitive.

Instead, we decide to make use of a Task History Table (THT) with a limited amount of tasks. When the table is full, we can replace old entries with new ones. Also, we decide to exploit hashing functions to store a compressed version of the data inputs. Data outputs have to be fully stored in the THT to allow task memoization.

Figure 1 shows the data structures added to the runtime system to enable task memoization and how they interact with the components already in the runtime. As mentioned in Section II-C, once dependences are satisfied, tasks (in the TDG) become ready for execution and are moved to the Ready Queue. When an idle thread pulls a task A from the RQ for execution, it first checks if this particular task can be memoized. This is done before actually executing the task itself, which adds some overhead in case of finally having to execute it. In order to minimize this overhead we do not compare the entire data inputs to identify task redundant executions, but only a hash key representing the inputs.

To check if task A can be memoized, the runtime accesses the THT. To index the table, we make use of a very precise hash key K_A using a subset of the inputs of task A. Section III-B explains in detail the mechanism to select the bits to generate the hash key. With the lower N bits of K_A, we index a THT with 2^N buckets, each containing M hash keys of the inputs of previously executed tasks, as well as their corresponding outputs. Each bucket is protected by a lock to support exclusive writes and allows parallel reads.

Next we check if one of these hash keys coincides with K_A. This is depicted in Figure 1 with the multiplexer that selects the hash keys in the bucket and compares them with K_A. If there is a task B with exactly the same hash key, then there is no need to execute task A. We just need to copy the outputs (depicted in the figure with the call to *copyOuts()*) from the matching task B and then release the bucket's lock.

The compiler translates to SIMD instructions both the copies of the outputs of B from the THT to the outputs of A and the copies of the outputs of a non memoized task to the THT. Those instructions are normally faster than all the individual writes to each output element performed otherwise if the task A was executed. The compiler might use SIMD instructions for the task itself (we have enabled vectorization for all the benchmarks presented in the paper), but the copies from and to the THT still are $10.75\times$ and $10.31\times$ faster, respectively, than actually executing the task (in our Experimental Setup; see Section IV for more details).

If none of the tasks has the same precise key, we check an additional smaller table, denoted In-flight Keys Table (IKT). This table simply stores at most as many hash keys as the number of threads exposed in the parallel execution. The IKT structure is needed only when the executions are performed with multiple threads because it maps the precise hash keys of the tasks that are currently being executed, i.e., in-flight. If a task B with the same hash key is found, the outputs cannot be immediately provided because task B is still executing. Instead the needed information is saved in task B's runtime-system data structures (*postponeCopyOuts()* in Figure 1), so that when it finishes, it copies its outputs to the outputs of task A. In our approach we allow multiple A-like tasks to store their petition for output copy in B-like in-flight task. We have found the IKT data structure to be

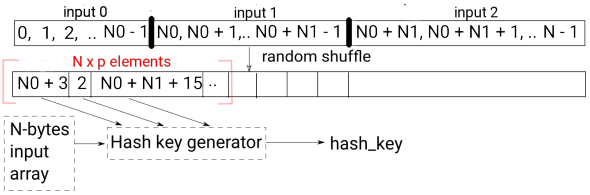


Figure 2. Hash key generation using the task data inputs.

a critical feature for some benchmarks that have very short reuse distances. Accesses to this structure are very fast when compared to the THT as they don't involve output copies. For this reason we protect the IKT with a single lock.

Finally, if a task A misses in the IKT, it has to be executed (after storing K_A in the IKT). When task A finishes, it retires K_A from the IKT and updates the THT by storing K_A together with the output in the corresponding bucket. If the bucket is full, the oldest task is evicted (first-in-first-out). This is depicted in Figure 1 as a call to *updateTHT&IKT()*.

B. Hash Key Generation

The generation of a precise hash key is an important component in the design of ATM. Depending on the quality of the generated key, we will have more or less potential for task memoization and approximation. The hash key computation time depends linearly on the size of the data inputs. To reduce this overhead, we decide to select a percentage $0 < p \leq 100\%$ of bytes in these data inputs.

To generate the hash key, we represent tasks' data inputs as a single vector of N bytes. Then we create a vector of N indexes that point to the vector with the concatenated inputs. Next, we randomly shuffle these indexes in a second vector and select the given percentage p of them. In particular, we select the first $N \times p$ indexes of the vector. In our implementation, we shuffle the vector of indexes the first time a task type is executed and store it in the runtime system. Consequently, in the following executions of the same task type, we will not need to recompute the shuffled vector of indexes, reducing its overhead. Other authors have followed a similar approach to select a subset of the bits of floating point operands [3].

The input bytes pointed by the selected indexes are passed to a hash key generator [12], which is known to give a collision once in 2^{32}. This number is larger than the number of tasks generated in any of our benchmarks. Finally, the generated hash key only requires 8 bytes of storage in the THT.

Figure 2 shows an example of the implemented mechanism to generate the hash key using the task data inputs. In this case, we assume a task with three data inputs, $input0$, $input1$ and $input2$, each with $N0$, $N1$ and $N2$ bytes, respectively, and a total of $N = N0+N1+N2$ bytes. After applying a random shuffle to the N bytes of the concatenated data inputs, we select the first $N \times p$ bytes to sample the input vector of bytes (marked with a red interval in the figure). The selected indexes are served to the hash key generator which will use them to compute the hash key.

Selecting an appropriate value of p is important for ATM as it controls the potential of task approximation. Smaller values of p increase the chances of approximating a task while reducing the overhead of generating the hash key. However, the approximated task could generate wrong outputs. In terms of precision, $p = 100\%$ provides the safest results and we denote it *Static ATM* in the rest of the paper. Section III-D describes an automatic approach to select an appropriate value of p, which we denote *Dynamic ATM*.

C. Type-aware Input Selection

When selecting a subset of the bytes of the data inputs, not all of them are equally representative. Upper bits of all data types tend to be more significant than the lower bits, and we can actually take this into account when we select input bytes. For example, in a double-precision floating point value, the most significant byte (MSB) contains the sign bit and 7 bits of the exponent, while the least significant byte contains the last 8 bits of the mantissa. This characteristic is commonly exploited by accelerator technology and low-power computing [3], [14], [16].

In order to select more representative bytes for the hash key generation, we extend our approach to consider the type of the data stored in a data input. With this goal, we have modified the shuffling function so that it first shuffles the indexes pointing to the MSBs of the data inputs. Then, the next MSBs are shuffled until we have shuffled the least significant bytes of the data inputs.

As a consequence, when selecting the window of $N \times p$ bytes in the data inputs, we will always choose first the MSB of each data type. In the example shown in Figure 2, if $input0$ and $input2$ contain integers (4-byte type) and $input1$ floats (4-byte type), for $p = 50\%$, 2 bytes out of 4 would be selected from each data type. This allows protecting the sign of the integer types and the most significant bits, while for floating point types, we first protect the sign and exponent and then the most significant bits of the mantissa.

To track dependences and manage the TDG of the application, the runtime system only requires the starting address of the data inputs and outputs and their size. To overcome this limitation, we have extended the runtime library API and modified the compiler to inform the runtime system about the types of the elements in each data input and output.

D. Automatic Task Approximation in the Runtime System

Next, we describe *Dynamic ATM*, our approach to automatically select an appropriate value of p to maximize task approximation without losing accuracy in the final results of the application.

In Dynamic ATM, we split the execution of the application into a *training* phase and a *steady-state* phase. The training phase explores different values of p and selects the

smallest value that preserves accuracy. In the training phase, the THT and IKT are updated as usual, while tasks are still executed to measure the accuracy of the task approximation. Once p has been determined, the steady-state phase starts, which should be much longer than the training phases to improve overall performance.

In the training phase, if task A can be approximated by a previously executed task B, we run task A anyway to measure the accuracy of the approximated task. Since the outputs of task B are stored in the THT, denoted x_{ATM}, we can measure the error when obtaining the outputs of task A, denoted $x_{correct}$. In our experiments, the Euclidean relative error (Equation 3) produces imprecise per-task error measurements that are not correlated with the overall program accuracy loss (note that the numerator and the denominator are accumulating floating point values).

As an alternative, we propose to use the Chebyshev relative error τ, shown in Equation 1. This metric does not suffer from precision problems because the reduction variable is not accumulating values, but selecting their maximum. In our experiments, this distance provides per-task errors well correlated with the overall correctness of the program. We consider that a task has been correctly approximated if τ is below a threshold τ_{max}.

$$\tau = \frac{max_{i=1}^{N}(|x_{correct_i} - x_{ATM_i}|)}{max_{i=1}^{N}(|x_{correct_i}|)} \quad (1)$$

In the training phase, we start with $p = 2^{-15} \cdot 100\%$ and every time we approximate a task and $\tau \geq \tau_{max}$, we double the value of p (15 possible configurations until we reach the maximum of $p = 100\%$). Once we find a number of tasks correctly approximated, denoted $L_{training}$, we select the current value of p for the steady-state phase. Section IV discusses how we selected the values of τ_{max} and $L_{training}$ for the evaluated applications. To support dynamic ATM, the structures THT and IKT have to be minimally extended: they require to store the value of p together with the hash keys as the value of p affects the hash key generation.

Additionally the adaptive algorithm identifies in the training phase the outputs of the tasks that exceed the maximum relative error (which are potentially related to chaotic behavior). The pointers of those outputs are stored in a C++ set and we do not memoize tasks whose output pointers are found in this set during the steady-state phase. This kind of accuracy control cannot be provided by a profiling technique if the training inputs are different from the test/production inputs because the output pointers that do not respond well to approximation might change when using a different input set. This feature is useful only for one of the applications we will present in Section IV-A.

E. Limitations

Since the THT and IKT structures store a hash key of the task inputs, it might happen that ATM suffers from a false positive. In our original approach, we implemented a hierarchy of hash functions with increasing computation requirements to avoid costly hash key computations for the tasks that would not benefit from ATM. We also implemented a final check (after all the hash keys have matched), where we were considering the complete inputs. However, the obtained results did not justify such a complex approach. Having a single hash key representing the selected inputs provides the best results in all the evaluated benchmarks. We did additional experiments to corroborate that no hash key collisions occur in the evaluated applications. As a consequence, no false positives are observed in the case of static ATM.

The second limitation of ATM has to do with the source code. The developer has to precisely express all the task data inputs and outputs. If a variable is modified by a task, but not specified in the data outputs (some programmers do that to reduce the overhead of tracking dependences), then task approximation will provide wrong results. Finally, task execution has to be deterministic, i.e. for a given input, the task will always produce the same output [1]. Tasks that make use of random values or access shared data via locks should not use ATM because the task output would depend on variables not explicitly indicated in the source code annotations.

To overcome this last limitation, we propose to rely on the developer of the application to identify the task types suitable for memoization. This can be achieved by extending the pragma annotations in OpenMP to let the programmer specify this feature. Although this approach requires the programmer to explicitly specify which task types are suitable for ATM, we believe that it is more reasonable than just memoizing tasks blindly. As described in Section IV, we have followed this approach for the evaluated applications.

IV. Experimental Setup

A. Applications and Platform

For our evaluation we present benchmarks from a wide set of domains: financial analysis, stencil-computation, machine-learning and linear-algebra.

Blackscholes calculates the prices for a portofolio of European options analytically with the Black-Scholes partial differential equation (PDE). There is a single task type in the benchmark and it is chosen for ATM.

Gauss-Seidel is a solver using the 2D Gauss Seidel five-point stencil computation. The matrix of the problem is divided into 2D blocks, each processed by a task. Neighboring columns and rows are obtained via copy-tasks. The

[1]This is the case of most applications with fully specified dependencies. Deterministic tasks enable transparent execution on accelerators [10], while the task data is automatically moved to and from a device chosen at runtime (e.g. GPU, FPGA, SMP). Furthermore, data-aware scheduling is enabled [4] which places a task on a computing element with fast and/or power efficient task data access.

Table I
BENCHMARKS DESCRIPTION.

Benchmark	Program Inputs	Task Inputs Size (bytes)	Task Inputs Types	Memoized Task Type	Number of tasks	Correctness Measured on
Blackscholes	Native input with 10 million options	393,216	float	bs_thread	6,109	Prices Vector
Gauss-Seidel	32×32 blocks of 1024×1024 elements	4,210,688	float	stencilComputation	20,480	Stencil Matrix
Jacobi	32×32 blocks of 1024×1024 elements	4,210,688	float	stencilComputation	20,480	Stencil Matrix
Kmeans	$2 * 10^6$ points, 16 centers, 100 dimensions	219,716	float, int	kmeans_calculate(distances)	39,063	Centers Vector
LU	20×20 blocks of 256×256 elements	786,432	float	bmod	670	$L * U - A$
Swaptions	Native with 512 swaptions	376	double	HJM_Swaption_Blocking	512	Prices Vector

Table II
DYNAMIC ATM PARAMETERS.

Benchmark	Black.	GS	Jacobi	Kmeans	LU	Swaptions
$L_{training}$	15	100	150	15	30	15
τ_{max}	1%	1%	1%	20%	1%	20%

Table III
ATM MEMORY OVERHEAD WITH RESPECT TO THE APPLICATION.

Blackscholes	GS	Jacobi	Kmeans	LU	Swaptions
4.9%	9.8%	9.26%	21.21%	7.7%	3.7%

kernel contains a random initialization of the blocks and the boundaries of the matrix emit heat at the same temperature. We choose the task type that computes the heat-diffusion for ATM, not the copy tasks.

Jacobi is a solver using the 2D Jacobi five-point stencil computation. The algorithm synchronizes at the end of each iteration, but there are no dependences between tasks in the same iteration. This application receives the same inputs as Gauss-Seidel. We choose the task type that computes the heat-diffusion for ATM as in Gauss-Seidel. Just calibrating the p in the training phase does not suffice to bound the accuracy losses of this application. A reduced set of task output pointers is responsible for this instability, which is identified by dynamic ATM in the training phase. As a result, a negligible number of tasks that make use of these pointers are not memoized.

Kmeans implements an unsupervised machine learning algorithm to cluster N d-dimensional points into k groups. The algorithm randomly distributes the points and selects k random initial centers. Then iteratively a task type (chosen for ATM) assigns a block of points to their closest centers. The centers are re-computed by a second task type.

LU decomposes a sparse $N \times N$ matrix A into a lower triangular matrix L and an upper triangular matrix U, such that $A = L \cdot U$. ATM is applied to the most frequently called routine, *bmod*, which subtracts the result of a row-column dot product from the elements of a vector.

Swaptions is an Intel RMS workload which uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions using Montecarlo Simulation instead of solving an analytical PDE. There is only one task type which is chosen for ATM.

Table I summarizes the main properties of the presented benchmarks. For an early discussion about the source of redundancy in those programs the reader may refer to Section V-D. Blackscholes and Swaptions are the only programs in the current version of the PARSECSs benchmark suite [8] with well defined task data inputs (all input and output data of the tasks is specified in the pragmas). We use a version of the LU [2] with configurable block size. Except LU and Swaptions, all benchmarks have thousands of tasks to train dynamic ATM. In the case of Swaptions, in order to have enough tasks for training dynamic ATM, we increase the size of the input from 128 to 512 swaptions. Finally, the presented benchmarks have very different size of task data inputs, ranging from hundreds of bytes (Swaptions) to over 4MB (Gauss-Seidel).

The pragma annotations also specify the amount of training tasks $L_{training}$ and the threshold τ_{max}, the per-task permitted inaccuracy with the Chebyshev error (see Equation 1). Table II summarizes those parameters. At least 15 training tasks should be used to allow Dynamic ATM to reach $p = 100\%$ and we observe that training ATM with 5% of the total amount of tasks is an appropriate empirical upper bound (average of 1.48%). The per-task inaccuracy may result in slightly different overall inaccuracy because of a known effect described by Sidiroglou et al. [19]. Applications may either magnify or diminish the impact of an approximated instance, a skipped loop iteration in their case, an approximated task in our case. In general, $\tau_{max} = 1\%$ provides good results, although relaxing this value (i.e., $\tau_{max} > 1\%$) provides further options to approximate tasks in Kmeans and Swaptions. Other benchmarks do not tolerate that well increasing the value of τ_{max}. Simple offline profiling techniques can help the application developer in finding both parameters.

All benchmarks are evaluated on an Intel E5–2670 Sandy Bridge-EP processor with 8 cores running at 2.6GHz and a shared last-level cache of 20MB. There are four 4GB DDR3 DIMMs running at 1.6GHz (a total memory of 16GB). All benchmarks are written in the OmpSs programming model [10], with task annotations equivalent to the ones supported by the OpenMP 4.0 specification [1]. We use the Nanos++ 0.9a runtime system and the Mercurium 1.99.9 source-to-source compiler, with gcc 4.8.2 as back-end compiler with autovectorization enabled.

B. ATM Sizing

ATM introduces overhead to the execution time, as well as on the memory footprint of the application depending on the dimensioning of the N and M parameters (defined in Section III-A) associated with the THT table. On the one hand parameter N must be large enough to avoid lock

contention in the THT. N=8 provides a 46% performance improvement with respect to N=0 (one hash bucket). Larger values of N do not significantly improve performance. On the other hand, most applications reach the maximum amount of approximated tasks at M=16, but kmeans needs a larger value of M=128 (this is the value used for all the experiments). Finally, ATM requires an average of 9.4% additional memory. For a more detailed, per benchmark breakdown of the memory costs see Table III.

C. Metrics

Speedup is computed from two executions with the same number of cores, one using ATM and the other not using it:

$$speedup = \frac{ExecutionTime_{no\_ATM}}{ExecutionTime_{ATM}} \quad (2)$$

.

Correctness. Given that the outcome of our programs is either a matrix or a vector, we can compute the relative error E_r between the output vectors or matrices with the Euclidean distance, as shown in Equation 3. The LU decomposition has an application-specific way of computing E_r, as shown in Equation 4.

$$E_r = \frac{\sum_{i=1}^{N}(x_{correct_i} - x_{ATM_i})^2}{\sum_{i=1}^{N}(x_{correct_i})^2} \quad (3)$$

$$E_r = \frac{|A - L * U|_2}{|A|_2} \quad (4)$$

Reuse. To measure task reuse, we make use of the percentage of memoized tasks thanks to ATM. ATM seeks to maximize reuse while maintaining the desired level of correctness.

V. Evaluation

A. Accuracy vs Performance Improvement

Figure 3 shows the performance speedup achieved by static ATM, dynamic ATM and two oracle configurations that execute with the best (smallest) p that guarantees a correctness above or equal to 95% and 100%. Static ATM uses all the input bytes for hash computation, while dynamic ATM automatically picks an appropriate percentage p of inputs. With Oracle (100%) if the p is small enough, the hash key computation overhead can be significantly reduced. Furthermore, with Oracle (95%), if the benchmark tolerates approximation, this configuration improves performance with respect to Oracle (100%) due to task approximation. This figure also shows the additional speedup that the IKT data structure provides with respect to just using the THT.

Figure 3 reveals the following observations:

Static ATM. Four out of six benchmarks benefit from exact task memoization. The speedups range between $1.07\times$ (Swaptions) and $5\times$ (Blackscholes). In contrast, Kmeans and Jacobi are not able to exploit task redundancy with this configuration. In the case of kmeans the centers change

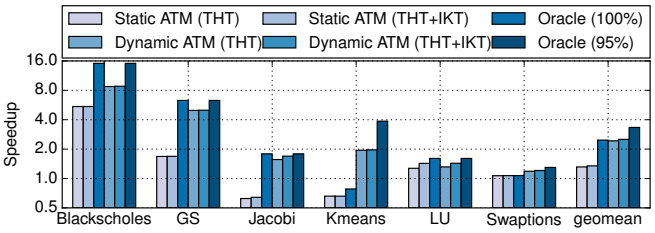


Figure 3. Speedup at a logarithmic scale with static and dynamic Approximate Task Memoization (ATM). Both approaches can make use of the Task History Table (THT) and the In-flight Key Table (IKT). The Oracle results are obtained with offline profiling and ensuring a 95% and 100% correctness in the final results of the application.

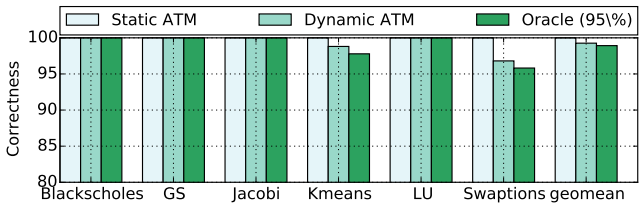


Figure 4. Correctness with static and dynamic Approximate Task Memoization (ATM). The Oracle results are obtained with offline profiling and ensuring a 95% correctness in the final results of the application.

in all the iterations, preventing exact memoization. In both cases hash computations and bookkeeping overheads lead to slowdowns. On average, static ATM achieves a $1.4\times$ speedup over the baseline.

Task approximation to reduce memoization overheads. In Figure 3, we see that Blackscholes, Gauss-Seidel, and Jacobi have a remarkable hash key computation overhead as shown by the significant difference between the static ATM and Oracle (100%). Those benchmarks raise their benefits from $5.5\times$ to $15.1\times$, from $1.68\times$ to $6.3\times$, and from $0.65\times$ to $1.73\times$, respectively. This suggests that task approximation can increase the memoization performance benefits with no added degradation in correctness. Also, it might be possible for the programmer to statically set a value of $p < 100\%$ but this would require more knowledge about the application (as the Oracle configurations). However, in this paper we rely on Dynamic ATM to choose the value of p, based on the specified error constraint τ_{max}.

Dynamic ATM. Automatically selecting a small percentage of the inputs to approximate tasks provides significant performance improvements. Dynamic ATM reaches an average $2.5\times$ speedup which is close to the average speedup of Oracle (95%). Moreover, all benchmarks benefit from dynamic ATM, with speedups betweeen $1.23\times$ (Swaptions) and $8.8\times$ (Blackscholes). However, the runtime system overhead of updating the THT prevents dynamic ATM to reach the potentially higher speedups of Blackscholes and Kmeans. The hash key computations are not responsible for the performance difference between Dynamic ATM and Oracle (95%) as they only represent 0.5% of the task duration in Dynamic ATM.

IKT. Adding an IKT to the runtime system makes Jacobi

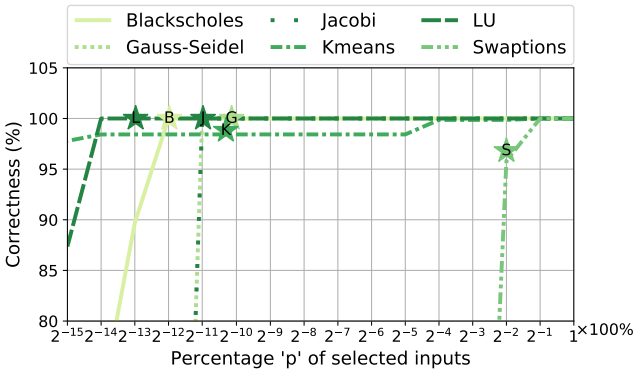


Figure 5. Correctness achieved with ATM depending on the percentage of selected inputs. The right-most point in the graph corresponds with the static ATM. The star symbol with the benchmark initial denotes the percentage selected in dynamic ATM. The curves of Jacobi and Gauss-Seidel overlap.

and LU increase their performance by 1.8% and 15% in static ATM, and by 13% and 12% in dynamic ATM. The IKT brings the speedup closer to the maximum achievable with the Oracle (95%) configuration.

Figure 4 summarizes the results obtained in terms of correctness. The Oracle (100%) results are not shown as they are always 100% correct, while the addition of IKT has a negligible effect in the final correctness. The first important conclusion from this figure is that static ATM always achieves a 100% correctness. This result shows that the design of the THT and the hashing function represent a robust way of performing memoization. Dynamic ATM is also able to achieve the 100% correctness for four benchmarks out of six. A small reduction in correctnes of 1.2% and 3.2% is obtained in the case of Kmeans and Swaptions, respectively. On average, dynamic ATM degrades correctness by only 0.7%, which is even a bit better than the correctness of Oracle (95%).

B. Sensitivity to the Percentage of Selected Inputs

Figure 5 shows the program overall correctness when running with a constant percentage p of selected inputs. We report the correctness of ATM with constant p between $2^{-15} \cdot 100\%$ and 100%. Note that the x-axis is in logarithmic scale. When decreasing the value of p, some benchmarks quickly degrade correctness. Swaptions already degrades correctness with $p = 12.5\%$. The other benchmarks tolerate values of p below 0.1% without degrading correctness.

Figure 5 also depicts the chosen configuration of dynamic ATM with a star symbol. As explained in Section III-D, the training phase begins with a $p = 2^{-15}$ corresponding to the left most x coordinate of the figure and jumps one step to the right (multiplies p by 2) each time a task does not meet the target error constraint τ_{max}. For all benchmarks, dynamic ATM selects a configuration that returns a correctness higher than 96.8%. LU and the stencil algorithms reach a correctness of either 100% or smaller than 90%, since errors can get easily propagated in those programs. Dynamic ATM

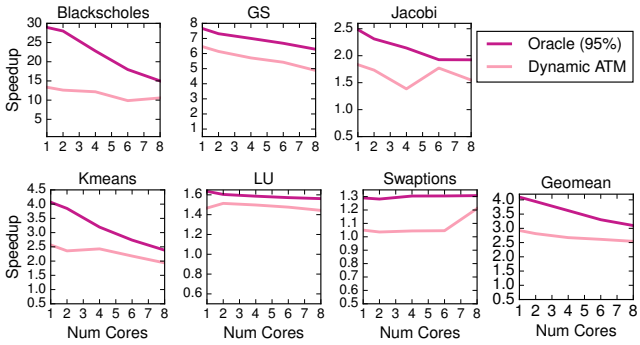


Figure 6. Performance speedups as we increase the number of cores from 1 to 8. Results are shown for Oracle (95%) and Dynamic ATM.

effectively identifies this situation. Most benchmarks do at most one additional step than necessary to ensure program correctness, demonstrating the effectiveness of the adaptive algorithm. Only Kmeans does five extra steps in the training phase, a conservative runtime decision that does not impact the final performance of the benchmark (it behaves very similarly for p between $2^{-14} \cdot 100\%$ and $2^{-5} \cdot 100\%$).

C. ATM Scalability

Figure 6 shows the speedup of dynamic ATM over different number of cores for the evaluated applications. The baseline corresponds to the parallel execution of the applications without using ATM. The x-axis shows the number of cores used in each experiment. We also include the speedups of Oracle (95%), which chooses the best configuration that guarantees a correctness above 95%.

Dynamic ATM adapts to large multicore scenarios. Figure 6 shows performance speedups between $3.0\times$ (1 core) and $2.5\times$ (8 cores), and the curve stabilizes (it is convex). When executing with more cores, some opportunities for task approximation may be missed. This is not a consequence of an increasing number of running tasks that hold reuse potential (as the number of cores grows) and whose output is not yet committed in the THT, given that we exploit this potential with the IKT structure.

However when using different number of cores, the order of task execution might change, affecting the election of the percentage p of task inputs for hash key computations. For instance, in Kmeans different results in the training phase make dynamic ATM choose in average a slightly more conservative value of p with 8 cores, namely 0.058%, rather than selecting 0.024% as in the single core configuration. This results in a slight decrease in the reuse percentage from 1 core to 8 cores. LU however, is the most affected benchmark by the reuse reduction. The percentage of reused tasks diminishes from 90% (1 core) to 49% (8 cores) because of an increase in p from 0.006% to 0.012%. Fortunately this different decision does not degrade significantly the speedup benefits at 8 cores because the tasks that dominate the execution time are still approximated. The opposite case happens in Swaptions: dynamic ATM chooses a more

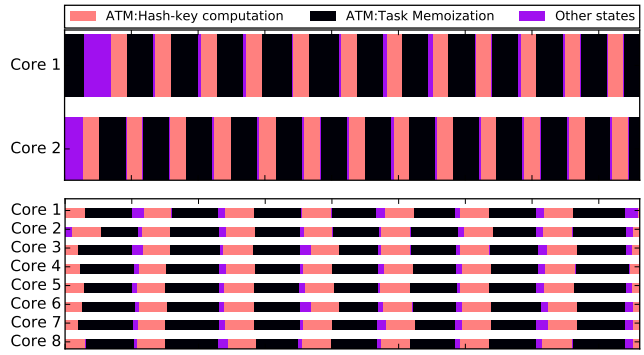


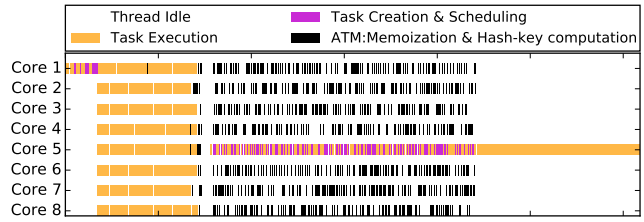
Figure 7. Gauss-Seidel execution trace with 2 (top) and 8 (bottom) cores using the Oracle (95%) approach. We zoom-in on a memoization intensive phase of the benchmark. Both traces have the same time scale. The legend uses the ATM prefix for the two states of interest related to Approximate Task Memoization. "Other states" covers all the other cases.

conservative percentage $p = 50\%$ at less than 8 cores and a $p = 25\%$ at 8 cores. This increases the percentage of approximated tasks from 5% to 17.6%.

Taking into account that task reuse remains stable when running the applications with the Oracle 95% configuration, we would expect that the achieved speedups to stay constant when increasing the number of cores. However Blackscholes, Gauss-Seidel, Jacobi and Kmeans have diminishing returns with the number of cores. In contrast, dynamic ATM, which is more conservative than Oracle (95%), has a better scalability. Next, we present a detailed analysis showing that the hardware and the runtime system limitations are responsible for those degradations:

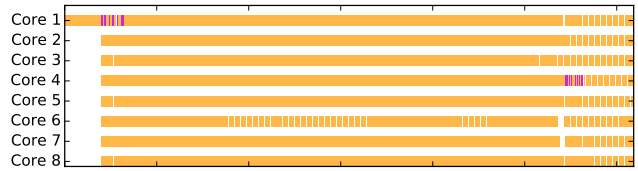
Hardware structure contention. Shared memory resources can be a limiting factor in some benchmarks especially at a small percentage p of inputs, in the program phases where lots of task reuse is being performed in parallel. Figure 7 shows two execution traces of Gauss-Seidel with 2 and 8 cores. Both traces make use of the same time scales (represented in the x-axis) and colors represent different thread states (there is one time line per available core). We observe that both states *ATM:Task memoization*, denoting the copies from the THT to the bypassed task's outputs, and *ATM:Hash-key computation* are on average 60% slower at 8 cores than at 2 cores. Given that the hash key computations and the copies from/to the THT are cache and main memory intensive activities, and that the traces clearly reflect that we are able to run those in parallel (the problem is not lock contention), we believe that the limitation comes from the contention in shared memory components such as the memory bandwidth.

Creation throughput. We observe that Blackscholes and Kmeans suffer from insufficient task creation throughput when increasing the number of cores. In Figures 8(a) and 8(c), we compare the execution of Blackscholes with and without dynamic ATM. We see that the task creation dominates more in the execution with dynamic ATM (Figure 8(a)). In fact, we confirm this limitation using Figures 8(b)



(a) Blackscholes with Dynamic ATM execution trace. The trace includes the execution of all the iterations in the program. Before the end of the first iteration, ATM has gathered enough information to start memoizing.

(b) Number of ready tasks with ATM. Same timeline as 8(a).



(c) Blackscholes without ATM execution trace. The first iteration is being shown and the start of the second one.

(d) Number of ready tasks without ATM. Same timeline as 8(c).

Figure 8. The same time scale is used in all the graphs. The execution with ATM executes all the iterations in less time than one complete iteration without ATM. "Thread Idle" also includes other states which represent less than 1% of the time

and 8(d) which show the number of ready tasks in the runtime system at each point of time. In Figure 8(c), tasks are created for a short amount of time (after each barrier) and then all the threads execute tasks. However in Figure 8(a) the master thread can not create enough tasks at the speed at which threads compute hash keys and approximate tasks. For this reason the ready queue is drained and stays empty. As soon as the master thread inserts a task, it is immediately retired by a worker thread which is able to memoize it in most cases. We have observed the same behavior in Kmeans.

Synchronization overheads of task memoization. Given the implementation of the per-bucket locks in the THT, the parallel reads per THT bucket capability and further analysis via execution traces suggests that the synchronization introduced by ATM does not represent a significant overhead in the execution time.

D. Source of Task Redundancy

This section identifies the sources of task redundant executions that allow ATM to improve performance in the evaluated applications. We find out that repetitive patterns in the application input sets contribute to this redundancy. Also, there is redundancy generated by the algorithm itself at runtime. Figure 9 summarizes this information for the evaluated applications with dynamic ATM. The x axis plots

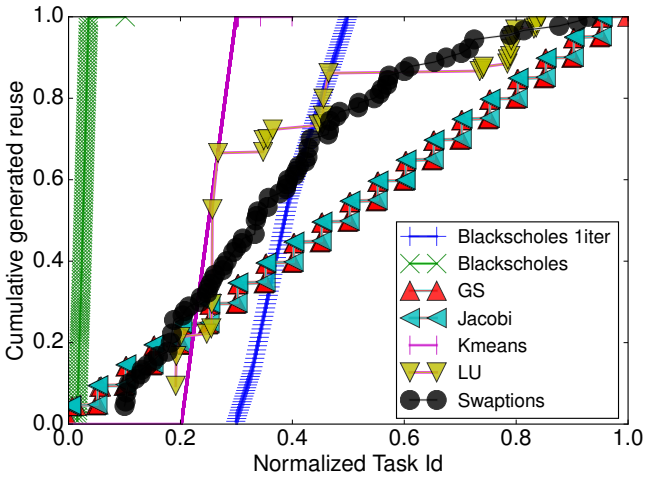


Figure 9. Redundancy generation during the execution of an application.

the normalized id of the executed tasks, whereas the non-normalized task id is a natural number which represents the order of task creation, and the y axis plots the cumulative reuse generated by those tasks. Each marker represents a task that contributes to increase task reuse. Therefore the first marker corresponds to the first task that will be reused.

Embarrassingly parallel algorithms such as Blackscholes and Swaptions have their redundancy in the program's inputs as this input is divided into blocks and processed independently. Most of the redundancy is generated early in the execution of Blackscholes (as shown in Figure 9), while in Swaptions it is spread during the whole execution history. We have detected that Blackscholes repeats the same algorithm multiple times, the last iterations being redundant, but even if we reduce the number of iterations to 1, we still see the same pattern but shifted to the right. The reuse when executing only one iteration is 50%. Swaptions also has program inputs redundancy, but it can be exploited at its full potential only with dynamic ATM, increasing the reuse to 20% (reuse is 7% for static ATM). Blackscholes and Swaptions are known to respond well to approximate techniques [19].

In the case of iterative algorithms, they also provide redundancy because the algorithm converges faster on some pieces of the domain. This is the case of Gauss-Seidel, Jacobi and Kmeans. The two stencil algorithms propagate heat towards the interior of a matrix representing a room. The temperature near the walls converges faster than in the interior, while many iterations are required to start changing the temperature in the center of the room. As a result, there are sub-blocks in the matrix performing redundant executions. We have also found redundancy in the initialization of the sub-blocks of the matrix due to the saturation of the random number generator. Figure 9 shows that the stencil algorithms generate redundancy during the whole execution and that some tasks produce results with higher reuse than others. Previous work has also found redundancy in stencil algorithms at finer granularities [23].

Kmeans iteratively computes the center of the clusters based on the points currently assigned to them. Some centers may converge before others because they have very well defined points belonging to the cluster. However the algorithm will keep computing the center of that well defined cluster, which is redundant. Effectively, Figure 9 shows that the first third of the tasks produce the most reused results. This redundancy is not in the inputs of the program, but it is generated during execution and it can only be exploited if we use task approximation. Other proposals also manage to perform approximation techniques on Kmeans [11].

Finally, in Figure 9 we can see that some tasks are more useful for redundancy elimination than others. This is especially important for LU, where we observe a steep slope with high distance between two tasks that generate reuse. In contrast, programs that generate redundancy at the beginning of the execution have tasks that provide their result to a very similar number of approximated tasks. LU is neither embarrassingly parallel nor an iterative algorithm, but it reuses at short distances and we see that this reuse spreads over the whole execution. We think this redundancy is both thanks to the algorithm and to the inputs. Figure 9 also justifies the need for continuously updating the THT, as redundancy appears spread over the whole execution for most of the benchmarks.

VI. Conclusion

This paper presents Approximate Task Memoization (ATM), a redundancy elimination technique at coarse granularities that boosts program performance by memoizing tasks that have equal or similar inputs to previously executed tasks. The main highlights of the paper are[2]:

Static ATM and ***Coarse Grained Memoization***. The paper presents static ATM, a technique that transparently reuses task results in the runtime system. Compared to the few proposals of coarse grained memoization, we employ our technique at higher granularities (up to 4MB) and we are not limited to chains of instructions. To avoid costly inputs comparisons, we only visit once the inputs of the tasks by implementing an efficient hashing mechanism. Without any approximation, static ATM provides a $1.4\times$ average speedup for the evaluated applications.

Dynamic ATM. We present dynamic ATM, an adaptive algorithm that automatically determines the appropriate value of p in order to run with good performance and bounding the accuracy loss. Dynamic ATM is able to find out configurations that provide a $2.5\times$ average speedup and accuracy losses below 3.2% (0.7% on average). No off-line profiling is required in such approach: the runtime system gathers all the needed information during execution. To measure task correctness, we discover that the Chebyshev

[2]All reported results are with respect to an 8-core baseline without ATM.

distance provides the best results, avoiding precision errors and being well correlated with overall program correctness.

ATM in a Real Multi-core. ATM implements a mechanism to cope with the artificial loss of redundancy when executing programs in parallel. In parallel executions, when a task becomes ready, other in-flight tasks run concurrently. These tasks may be useful for memoization although they have not committed their results to the THT. For this reason, ATM also includes the IKT data structure for in-flight task memoization. Additionally, implementing ATM in the runtime system for multi-core processors involves careful lock design in the memoization mechanism, and taking into account micro-architectural shared resources. With ATM we obtain such dramatic performance improvements that we reach peak memory system utilization and maximum runtime system task creation throughput. Despite those limitations the dynamic ATM is able to scale well in the ranges of 1 to 8 cores.

ACKNOWLEDGMENTS

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the Spanish Government (grant SEV2015-0493 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and the European HiPEAC Network of Excellence. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243). I. Brumar has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2015/12849.

REFERENCES

[1] OpenMP: Application program interface. 2013.

[2] Taskified LU with constant block size. https://pm.bsc.es/projects/bar/attachment/wiki/SparseLU/, Feb. 2017.

[3] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54:922–927, 2005.

[4] L. Alvarez, M. Moretó, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguadé, and M. Valero. Runtime-guided management of scratchpad memories in multicore architectures. In *PACT*, pages 379–391, 2015.

[5] S. Arora, D. Karger, and M. Karpinski. Polynomial time approximation schemes for dense instances of np-hard problems. In *STOC*, pages 284–293, 1995.

[6] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *CGO*, pages 235–245, 2015.

[7] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, et al. Runtime-aware architectures. In *EuroPAR*, pages 16–27, 2015.

[8] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero. PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. *ACM Trans. Archit. Code Optim.*, pages 41:1–41:22, 2016.

[9] D. Conners and W.-M. W. Hwu. Compiler-directed dynamic computation reuse:rationale and initial results. In *MICRO*, pages 158–169, 1999.

[10] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, pages 173–193, 2011.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, pages 449–460, 2012.

[12] B. Jenkins. A hash function for hash table lookup. http://www.burtleburtle.net/bob/hash/doobs.html, Sept. 2016.

[13] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS*, pages 138–147, 1996.

[14] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, pages 369–381, 2015.

[15] J. S. Miguel, M. Badr, and N. E. Jerger. Load value approximation. In *MICRO*, pages 127–139, 2014.

[16] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: balancing efficiency and flexibility in specialized computing. In *ISCA*, pages 24–35, 2013.

[17] S. Sastry, R. Bodik, and J. Smith. Characterizing coarse-grained reuse of computation. In *FDDO*, page 274, 2000.

[18] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO*, pages 248–258, 1997.

[19] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT*, pages 124–134, 2011.

[20] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *ISCA*, 1997.

[21] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *MICRO*, pages 205–215, 1998.

[22] M. Valero, M. Moretó, M. Casas, E. Ayguadé, and J. Labarta. Runtime-aware architectures: A first approach. *Int. J. Supercomputing Frontiers and Innovations*, (1):29–44, 2014.

[23] S. Wen, X. Liu, and M. Chabbi. Runtime value numbering: A profiling technique to pinpoint redundant computations. In *PACT*, pages 254–265, 2015.