Performance Portability Across Heterogeneous SoCs Using a Generalized Library-Based Approach

SHUANGDE FANG, ZIDONG DU, YUNTAN FANG, and YUANJIE HUANG,
SKL Computer Architecture, ICT, CAS, China; Graduate School, CAS, China
YANG CHEN, Microsoft Research, China
LIEVEN EECKHOUT, Ghent University, Belgium
OLIVIER TEMAM, INRIA, Saclay, France
HUAWEI LI, YUNJI CHEN, and CHENGYONG WU, SKL Computer Architecture, ICT, CAS, China

Because of tight power and energy constraints, industry is progressively shifting toward *heterogeneous* system-on-chip (SoC) architectures composed of a mix of general-purpose cores along with a number of accelerators. However, such SoC architectures can be very challenging to efficiently program for the vast majority of programmers, due to numerous programming approaches and languages. Libraries, on the other hand, provide a simple way to let programmers take advantage of complex architectures, which does not require programmers to acquire new accelerator-specific or domain-specific languages. Increasingly, library-based, also called algorithm-centric, programming approaches propose to generalize the usage of libraries and to compose programs around these libraries, instead of using libraries as mere complements.

In this article, we present a software framework for achieving *performance portability* by leveraging a generalized library-based approach. Inspired by the notion of a component, as employed in software engineering and HW/SW codesign, we advocate nonexpert programmers to write simple wrapper code around existing libraries to provide simple but necessary semantic information to the runtime. To achieve performance portability, the runtime employs machine learning (simulated annealing) to select the most appropriate accelerator and its parameters for a given algorithm. This selection factors in the possibly complex composition of algorithms used in the application, the communication among the various accelerators, and the tradeoff between different objectives (i.e., accuracy, performance, and energy).

Using a set of benchmarks run on a real heterogeneous SoC composed of a multicore processor and a GPU, we show that the runtime overhead is fairly small at 5.1% for the GPU and 6.4% for the multi-core. We then apply our accelerator selection approach to a simulated SoC platform containing multiple inexact accelerators. We show that accelerator selection together with hardware parameter tuning achieves an average 46.2% energy reduction and a speedup of 2.1× while meeting the desired application error target.

Categories and Subject Descriptors: D.3.4 [**Software: Programming Languages**]: Processors—*Optimization, Runtime Environments*

General Terms: Design, Performance, Languages, Experimentation

Additional Key Words and Phrases: SoC, heterogeneity, performance portability, library-based programming, approximate computing

New article, not an extension of a conference paper.
Authors' addresses: S. Fang (corresponding author), Z. Du, Y. Fang, Y. Huang, H. Li, Y. Chen, and C. Wu, Institute of Computing Technology, Chinese Academy of Sciences, No 6. Kexueyuan South Road, Haidian District, Beijing, China; email: fangshuangde@ict.ac.cn; Y. Chen, Microsoft Research, Beijing, China; L. Eeckhout, Department of Electronics and Information Systems, Ghent University, Belgium; O. Temam, Inria Saclay, France.

DOI: http://dx.doi.org/10.1145/2608253

ACM Reference Format:
Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. 2014. Performance portability across heterogeneous SoCs using a generalized library-based approach. ACM Trans. Architec. Code Optim. 11, 2, Article 21 (June 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2608253

1. INTRODUCTION

In order to achieve high performance at low power and energy, the industry is evolving toward system-on-chip (SoC) architectures, composed of a heterogeneous mix of general-purpose cores and special-purpose accelerators [Esmaeilzadeh et al. 2011] (see Figure 1). The accelerators can span a broad range of architecture designs: from simple cores to DSPs, GPUs, FPGAs, and ASICs. While programming multicores is already a challenging task, programming a heterogeneous SoC multicore architecture with accelerators is even more complex. Each accelerator often comes with its own language and specific complications: manual memory management in CUDA for GPUs, RTL-level description for FPGAs, specific APIs for ASICs and DSPs, and so forth. Moreover, in future high-performance architectures, one may expect a combination of such accelerators since each usually targets a different application domain. To further complicate the programming challenge, as these heterogeneous SoCs evolve in the future, the mix and nature of accelerators are likely to change over time. As a result, the key programming challenge ahead relates to not only how to program these architectures but also how to achieve good performance across (generations of) architectures without requiring a software rewrite, a property called *performance portability*.

Several environments have been proposed to facilitate the task of writing programs for heterogeneous SoCs, either by offering a single address space [Lin et al. 2012; Diamos 2008], APIs for managing multiple versions of the same task for different accelerators [Diamos 2008; Augonnet et al. 2009], OpenMP-like annotations to identify target accelerators [Dolbeau et al. 2007], or domain-specific languages from which efficient accelerator code can be generated [Brown et al. 2011]. However, all these methods require the programmer to be deeply involved in the task of writing programs for heterogeneous SoCs. While this is perfectly acceptable for programmers with a solid architecture background and willing to learn esoteric programming approaches, it is not for the vast majority of programmers, because of either skill or time limitations.

A simple and attractive solution for letting nonexpert programmers take advantage of accelerators is to rely on *libraries*. Since both libraries and accelerators essentially implement *algorithms*, libraries come across as the natural programming abstraction for exposing accelerators to the programmer—a technique called *library-based* or *algorithm-centric* programming [Phothilimthana et al. 2013]. Composing a program out of different libraries also improves programming productivity by reducing the role of the programmer to producing glue code around library calls. More importantly, it makes a program *performance portable*, that is, easily portable and run efficiently across a broad set of architectures, by swapping the library implementing an algorithm for a given accelerator with a library implementing the same algorithm for another accelerator. Using a runtime to dynamically select accelerators can provide performance portability in a way that is transparent to the user.

In this article, we present a software framework for achieving *performance portability* on heterogeneous SoC architectures using a generalized library-based approach. The library consists of a collection of algorithms along with various implementations for different SoC components (multicore and accelerators such as GPU, DSP, FPGA, etc.). Each algorithm is presented to the programmer as a component—a concept borrowed from software engineering and HW/SW codesign [Cesário et al. 2002; Arató et al. 2005;

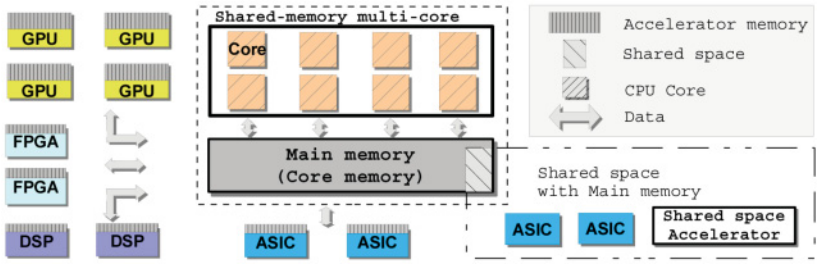

Fig. 1. Example heterogeneous SoC architecture.

Cheung et al. 2007]—using wrapper code to enable the runtime to dynamically select the most appropriate implementation on the most appropriate accelerator within a given execution context.

The key principles of the proposed framework are the following. First, the programmer decomposes his or her program into a set of algorithms. The programmer can either implement the algorithms him- or herself or simply select existing libraries already implementing them, only adding the algorithm-level semantic information. Whether the implementation corresponds to C code meant to be run on a single core, multithreaded code for a multicore, CUDA code for a GPU, Verilog code for an FPGA/CGRA, or even a simple call to an ASIC is of little importance to the programmer. The only task for the programmer is to write wrapper code that transfers input data from main memory to the accelerator and transfers output data from the accelerator to main memory. The main program calls these wrappers and initiates the transfers to/from the accelerators; running the library code on the accelerators is handled by the runtime. Second, and more importantly, the runtime is in charge of dynamically selecting the most appropriate accelerator and its hardware parameters for each algorithm in the application given the execution context. In order to achieve performance portability in an automated way that is transparent to the end-user, our runtime employs simulated annealing to dynamically explore the best possible online schedule while managing exploration overhead. The dynamic selection factors in the possibly complex composition of algorithms used in the application, the communication among the various accelerators, and the complex interplay between various optimization criteria such as application accuracy, performance, and energy consumption.

We first evaluate the overhead of the runtime and the wrapping approach using a set of six benchmarks on an existing heterogeneous SoC. We show that the execution time overhead of our approach is small, at 5.1% on average for GPUs and 6.4% for multicores. We subsequently consider both a more complex and more realistic scenario by simulating an SoC platform with four different architectures (a CPU, a GPU, a CGRA, and an ASIC). We provide an even richer mix of time, energy, and accuracy goals by augmenting the accelerators with the ability to perform approximate computations [Sampson et al. 2011; Esmaeilzadeh et al. 2012a], and we let the runtime find the best accelerator combinations and their hardware parameters for a given set of user-defined application targets. Our experimental results report that the runtime can achieve 46.2% energy reduction on average and a speedup of 2.1× on average while still meeting the desired application error targets.

Overall, we make the following contributions in this article:

—We propose a practical approach to program emerging heterogeneous SoC architectures. It integrates the ideas of swappable components, wrappers, and semantic annotations in such a way that nonexpert programmers can still write programs in

the traditional von Neumann model using existing libraries while at the same time enabling the runtime to automatically adapt applications to run efficiently across different heterogeneous SoCs, thereby achieving performance portability.
—We design and implement an intelligent runtime to support our generalized library-based programming approach. It employs simulated annealing coupled with a budget-based strategy to automatically explore the best possible scheduling scheme while at the same time managing the benefits and exploration overhead.
—We demonstrate our approach by componentizing six benchmarks, with the components implemented by wrapping existing libraries. The results show only 5.1% overhead for GPUs and 6.4% for multicores. We evaluate our approach on a simulated SoC platform with four different inexact accelerators, using 20,000 unique datasets per benchmark. The results show we can achieve 46.2% energy reduction and 2.1× speedup on average compared to a general-purpose multicore processor.

In Section 2, we present the overall programming approach and the runtime. In Section 3, we measure the overhead and evaluate the approach on a real platform using GPUs and multicores. Then, in Section 4, we apply it to a simulated SoC with several accelerators. In Section 5, we discuss related work, and we present our conclusions in Section 6.

2. PRINCIPLES

Component-based programming is quite well known to many areas. In order to cope with the increasing complexity of system development, software engineering advocates build ing programs using existing software components [Heineman and Councill 2001; Oberleitner and Gschwind 2002]. In HW/SW codesign, the component-based design flow allows designers to work at a very high level of abstraction, and the wrapper or bridge component is proposed to construct a unified software interface around the hardware IP blocks [Cesário et al. 2002; Arató et al. 2005]. In this study, we use the notion of swappable components to help nonexpert programmers leverage existing accelerator libraries to achieve performance portability on heterogeneous SoC architectures.

We provide two machine models in the programming approach: one for the main program and a second one for the libraries. The main program can be viewed as *glue code* calling components. And we let nonexpert programmers write simple wrappers to encapsulate the existing libraries into components using the second machine model. Finally, our runtime will efficiently merge the two models and yield the best possible component scheduling scheme to achieve performance portability. In the next sections, we first elaborate on how we organize the program using the two machine models, and we then describe how it can be realized dynamically through a runtime.

2.1. Library-Based Programming

We first describe the two machine models corresponding to the main code and to library/wrapper code for accelerators. From now on, we also refer to library/wrapper code as a software *component*.

2.1.1. Main Code. The programmer is exposed to the classical von Neumann machine model composed of a processor and memory (see Figure 2). The program can be written using traditional programming languages, such as C/C++. The main difference with traditional programming approaches is that the programmer decomposes the program into a set of tasks corresponding to known algorithms and annotates the code to specify the nature of the algorithms and the semantics of the algorithm interfaces.

Consider JPEG image compression as an example, which is described in Figure 3. The image is decomposed into 8×8 image subblocks, on each of which four steps are performed: *Colorspace Conversion* (CC), *Discrete Cosine Transform* (DCT),

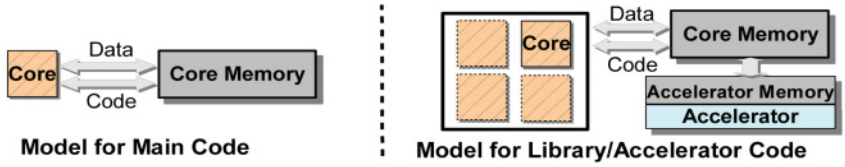

Fig. 2. Machine model for the main program (on the left) versus the machine model viewed by the libraries/accelerators (on the right).

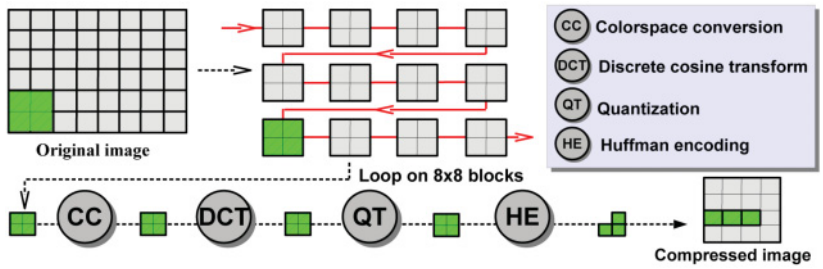

Fig. 3. JPEG compression steps.

```
1  // ... Include headers ...
2  // ... Variables definitions ...
3  @APP_ACCURACY_METRIC: SSIM
4  // Function definition of SSIM ...
5  @APP_ACCURACY_BOUND: 0.95; @APP_ENERGY_WEIGHT: 0.5; @APP_SPEEDUP_WEIGHT: 0.5;
6  int main(int argc, char *argv[]) {
7      // ... Initialization ...
8      for (int yhpos = 0; yhpos < height; yhpos += 8 ) {
9          for (int xhpos = 0; xhpos < width; xhpos += 8 ) {
10             imageBlocks = getImageBlock(imageBuffer, xhpos, yhpos);
11             /* Colorspace conversion: R G B to Y Cb Cr color channels */
12             @CPT_CALL ("Colorspace_conversion", imageBlocks, 8, 8,
13                        Y_buffer, Cb_buffer, Cr_buffer);
14             /* Call DCT, QT and Huffman encode for Y, Cb and Cr color channels */
15             @CPT_CALL ("DCT", Y_buffer, YDU_DCT_buffer, 8, 8);
16             // ... Call DCT for Cb and Cr color channel ...
17             @CPT_CALL ("QT", YDU_DCT_buffer, YQT_buffer, 8, 8);
18             // ... Call QT for Cb and Cr color channel ...
19             @CPT_CALL ("Huffman_encode", YQT_buffer, 8, 8, YOut_buffer,
20                        &YOut_len, YDC_HuffmanTable, YAC_HuffmanTable, ...);
21             // ... Call Huffman encoding for Cb and Cr color channel ...
22         }
23     }
24   return 0;
25 }
```

Fig. 4. Main code for JPEG.

Quantization (QT), and *Huffman encoding* (Huffman). The corresponding main code is shown in Figure 4. DCT takes a subblock of the image, applies a discrete cosine transform, and outputs a modified block of the same size. The programmer identifies that this task is a candidate for accelerator execution by using an annotation in the form of a special function called @*CPT_CALL* (which stands for *component call*). The first argument of the call is a string identifying the algorithm, and the remaining arguments are the component arguments in a predefined order matching the specification of the algorithm implementation.

For library developers, or in case the programmer decides to write a custom version of his or her algorithm, we provide a set of annotations for exposing the appropriate

```
1  /* Discrete Cosine Transform */
2  @CPT_BEGIN
3  @CPT_NAME: DCT
4  @CPT_INTERFACE:
5  Y_buffer (in, float*),
6  YDU_DCT_buffer (out, signed short int*),
7  width (in, unsigned long int),
8  height (in, unsigned long int)
9  @CPT_VERSION: CPU | INEXACT_CORE | CGRA
10   /* .. DCT CPU/Inexact CPU code .. */
11   for each 8x8 image block {...}
12 @CPT_VERSION: GPU
13   /* .. DCT CUDA code .. */
14   // Init (CUDA, accelerator)
15   // Transfer data from CPU to GPU
16   dct_kernel(d_Ybuffer, d_YDU_DCTbuffer,
17   ...)
18   // Transfer data from GPU to CPU
19 @CPT_END
```

Fig. 5. Annotations.

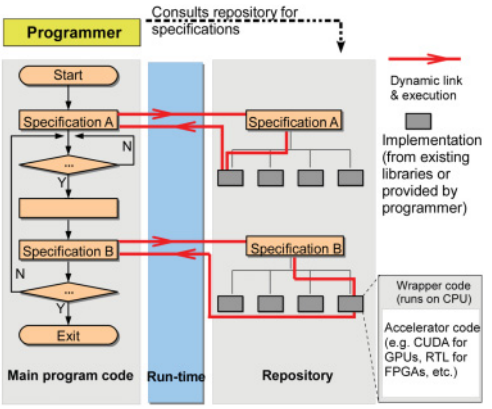

Fig. 6. Program flow.

semantic information to the runtime (see Figure 5). These annotations essentially specify the algorithm, the implementation interface (inputs and outputs), and the target accelerator(s).

2.1.2. Repository. The software components containing the library versions of algorithms reside in a repository that is accessed by the runtime to find matching (equivalent) libraries (see Figure 6). The programmer consults this repository to determine which algorithms can be instantiated along with their interface specifications and adds libraries (and possibly wrappers if they are generic enough) to the repository. This approach factors in the fact that algorithms may have many different variants, each with different properties, and it organizes algorithms in a graph describing the relationship between algorithm variants based on these properties. Note that there is no need to embed the repository within the runtime, nor to have a single such repository. There is only a need to standardize the *wrapper* around accelerator code.

2.1.3. Accelerator Code. The wrapper is designed for the generic machine model presented in Figure 2. The accelerator is assumed to be part of a heterogeneous SoC, including at least one general-purpose CPU, so there is at least one core on which the wrapper can be executed. The core communicates with memory (either main memory of a shared-memory multicore or the local memory associated with that core within a distributed multicore), which we call the *core memory*. The accelerator itself has its own local memory, called the *accelerator memory*. Note that this accelerator memory is optional, since some accelerators can access the shared memory address space. This simple model is compatible with almost any existing accelerator, including GPUs, FPGAs, ASICs, and DSPs. The main distinction between these accelerators is the way they load/store data to/from the accelerator, and that is the main role of the wrapper.

Let us again consider the DCT example of JPEG. The DCT specification stipulates that a subblock of the image is passed as a pointer, together with the subblock square dimension and the original image row dimension. Assume that a programmer has written a wrapper for a CUDA version of the DCT; that is, the wrapper will contain the code for transferring data from the core memory to the accelerator memory and back. The wrapper code is shown in Figure 7. Note that there is one more argument than in the *@CPT_CALL*, namely, the accelerator ID, which is passed to the wrapper by the runtime, and which indicates the accelerator on which the code will be executed.

```
#pragma Device_Requirements: 1 GPU, #cores >= 256, memory >= 512
void DCT_CUDA(float *ImgSrcF, float *DU_DCT_buffer,
              unsigned short int width, unsigned short int height,
              Accelerator accelerator_id) {
    cudaSetDevice(accelerator_id);
    /* Allocate accelerator memory */
    float *DevMem_inBuff, *DevMem_outBuff;
    cudaMalloc(&DevMem_inBuff, width * height * sizeof(float));
    cudaMalloc(&DevMem_outBuff, width * height * sizeof(float));
    /* Copy data from main memory to accelerator memory */
    cudaMemcpy(DevMem_inBuff, ImgSrcF, width, height, cudaMemcpyHostToDevice);
    // ... Setup execution parameters: GridFullWarps, ...
    /* Load and execute DCT kernel */
    CUDAkernelDCT<<<GridFullWarps, ThreadsFullWarps, SharedMemAmount>>>
                (DevMem_inBuff, DevMem_outBuff, width, height);
    /* Copy results back from accelerator memory to main memory */
    cudaMemcpy(DU_DCT_buffer, DevMem_outBuff, width, height, cudaMemcpyDeviceToHost);
    /* Cleanup */
    cudaFree(DevMem_inBuff);  cudaFree(DevMem_outBuff);
}
```

Fig. 7. Library wrapper code for the CUDA version of DCT.

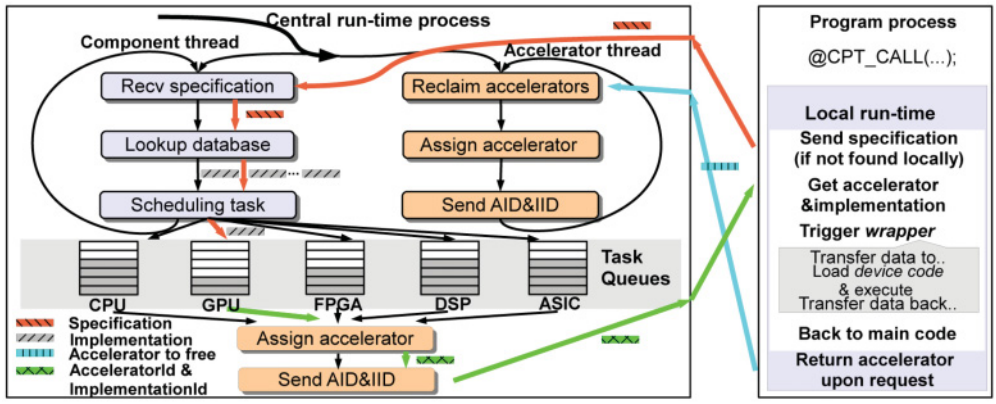

Fig. 8. Runtime.

The wrapper also includes a machine description pragma stipulating with which accelerator(s) it is compatible. The environment provides a standardized syntax for describing accelerators. Besides bridging accelerators and cores, wrappers fulfill another important role, namely, data structure compatibility. It may not always be the case that the data structures used in the different libraries are compatible. In such cases, the memory copy operation occurring in the wrapper is an opportunity to convert data structures. The ability to do data structure transformations (and not just copies) within wrappers can potentially expand the scope of libraries.

2.2. Combining the Two Machine Models With a Runtime

The main role of the runtime is to bring together the main (i.e., "glue") code and the accelerator codes, and to do so efficiently enough so that the performance benefits of libraries and accelerators are not wiped out by the overhead of copying occurring within wrappers.

2.2.1. Linking Main and Accelerator Code. The runtime is composed of two main threads: the *accelerator* thread and the *component* thread (see Figure 8).

The accelerator thread loops over the following three main functions: (1) check which accelerators are available, (2) submit tasks to accelerator queues (there is one queue per

```
1 @CPT_CALL(component_name, cpt_arg1, cpt_arg2 ...) {
2     // Inform the local run-time of the component_name
3     // Wait and get the component implementation with the assigned accelerator
4     // Trigger the wrapper to load and execute accelerator code on the assigned accelerator
5     // Return accelerator upon request
6 }
```

Fig. 9. Pseudo-code of the *@CPT_CALL* routine.

accelerator type), and (3) provide the accelerator (and component code) ID to the main code so that it can run the algorithm on the accelerator. The component thread loops over the following three main functions: (1) receive the component specification from the main process, (2) look up the repository for compatible accelerator implementations, and (3) select one implementation (based on the available accelerators) and schedule the execution in the corresponding accelerator queue.

Let us now briefly run through the execution of one call. A *@CPT_CALL* in the main code actually results in a call to the runtime. The runtime then consults the repository(ies). These repositories are cached in a local database, embedded with the runtime, for fast access. (Note that the cache is periodically refreshed by an independent process, not on the critical execution path.) Based on the available accelerators and on the machine description of the matching components in the database, the runtime selects an accelerator on which to execute, as well as the associated wrapper. If implementations corresponding to several accelerator types are available, the runtime factors in the queue occupancy. As soon as an accelerator has completed its task, code embedded in *@CPT_CALL* will notify the runtime, which can assign the next pending task from the corresponding queue. The corresponding wrapper code is passed to the main code through interprocess communication. As soon as it is received, the general-purpose CPU executes the wrapper code, which, in turn, transfers the data from the core memory to the accelerator memory.

Dynamic loading of accelerator code. The wrapper and accelerator codes are not defined at compile time in the main code; they can vary, possibly at every call, based on the occupancy of the various compatible accelerators. As a result, wrappers are implemented as dynamically linked files (`.so` in Linux or `.dll` in Windows) as *Position Independent Code* (PIC), which means that they can be loaded into any program's address space during execution. This dynamic linking results in performance overhead, which we quantify in our experiments, as described later in the article.

2.2.2. Distributed Runtime for Efficient Management of Accelerators. Besides the aforementioned PIC, the runtime introduces execution time overhead due to accelerator/component selection and because of communication between the accelerator(s) and the main process.

In order to tackle both issues, the runtime is implemented in a distributed manner. Unlike *Harmony* [Diamos 2008] or *Reflex* [Lin et al. 2012], the runtime is not distributed among (executed on) the accelerators themselves, which would require modifying the library code, but among the different general-purpose cores, each executing one program (several can be executing simultaneously running independent programs). This is done transparently to the programmer via the *@CPT_CALL* routines.

The runtime is split in a *central* runtime process and multiple *local* runtimes, which are embedded in the main code via the *@CPT_CALL* routines (see Figures 8 and 9). The central runtime contains the accelerator queues and has a full view of accelerator occupancy. However, it can temporarily delegate the management of an accelerator to a local runtime. The process actually interacts only with the local runtime. If the local runtime contains the accelerator code and owns a compatible accelerator, it can

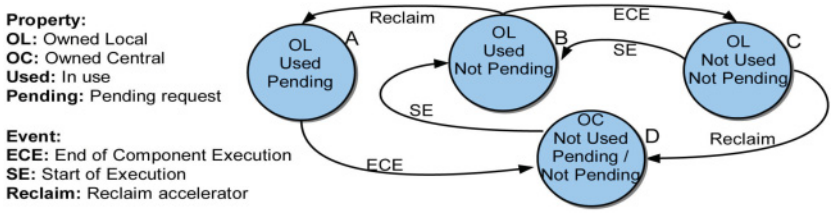

Fig. 10. Accelerator state automaton.

immediately assign it to the main code. As a result, it can avoid both overheads: central runtime selection process and long interprocess communications. Still, it may happen that the central runtime needs to reclaim the accelerator for another process, so the local runtime maintains a state of the delegated accelerators, which the central runtime can independently (and remotely) update via a hand-shaking process. The overall management of accelerators by the central and local runtimes is described in more detail later and in Figure 10.

There are four states in total, *A* through *D*, and four properties that define the states for accelerator management: *OL (Owned Local)* means that the current accelerator is owned by the local runtime; *OC (Owned Central)* means that the current accelerator is owned by the central runtime; *Used* means the accelerator is currently used; and *Pending* means there is a pending request from the central runtime for this accelerator. State *A* means that the accelerator is owned and currently used by the local runtime, and there is a request pending; the other states are defined as described in Figure 10. There are three events that will trigger the transitions among the states: *ECE (End of Component Execution)* means the end of a component execution on the accelerator; *SE (Start of Execution)* means that a component starts executing on an accelerator; and *Reclaim* means the central runtime needs to use the accelerator.

If the accelerator is in state *B*, that is, owned by the local runtime (*OL*), and there is no pending request from the central runtime (see Figure 10), upon completion (*ECE*), it transitions to state *C*. If a central runtime request arrives during execution (i.e., *Reclaim*) while in state *B*, it transitions to state *A*; upon completion (*ECE*), it then returns ownership of the accelerator to the central runtime and transitions to state *D*. If it is in state *C* and no central runtime request arrives, but the accelerator is no longer needed by the local runtime, it also becomes owned by the central runtime (*OC*) and transitions to state *D*. If the central runtime owns an accelerator (state *D*) and receives a request from the local runtime, upon beginning of execution (*SE*), the accelerator is delegated to the local runtime, and it transitions from state *D* to *B*.

3. PERFORMANCE PORTABILITY AND OVERHEAD ON A REAL PLATFORM

In this section, we quantify (i) how the library-based programming approach can deliver performance portability and (ii) the overhead introduced by the runtime. We do so on a real hardware platform consisting of a general-purpose multicore CPU along with a GPU.

3.1. Methodology

Platform. We consider a four-core 2.40GHz Intel(R) Xeon(R) (E5620 family) processor, with a 4 × 3MB shared L2 cache and 12GB of main memory. We use SUSE Linux Enterprise Server release 11 (SP1) with a 2.6.32.12 kernel; we use `gcc` version 4.4.6. The GPU card is an NVidia Tesla C2050, which has 448 stream processors running at 1.15GHz attached to a 3GB GDDR5 memory with a bandwidth of 144GB/s; the NVidia GPU driver version is 285.05.33, and we use `nvcc` 4.0 for GPU code generation.

Benchmarks. We use six benchmarks: `LEUKOCYTE`, `NW`, `BACKPROP`, `STREAMCLUSTER`, `KMEANS`, and `JPEG`. The first five are extracted from the Rodinia benchmark suite [Che et al. 2009], covering pattern recognition, medical imaging, bioinformatics, and data mining. We extracted multicore and GPU component codes from the Rodinia benchmarks. `JPEG` is our own custom implementation of JPEG image compression, which is about 10% faster than `libjpeg` [Lane and the Independent JPEG Group 1991] on a single core; we also provide multithreaded and GPU versions of the JPEG components. In Table I, we briefly describe the benchmark algorithms and datasets; we also report the percentage of the total execution time spent in the algorithms, and which accelerator version is available for each algorithm.

Measurements. We use the Time Stamp Counter (TSC) hardware cycle counter [Intel 2011] to time (whole or part of) program executions. We repeat each experiment 30 times and report the average results (arithmetic mean for times, harmonic mean for speedups). The speedups reported are relative to the execution time of the original program (before componentization) compiled with `gcc -O3`. The runtime implementation also requires measuring the time spent in interprocess communication (between the central and local runtimes): this is done by measuring round-trip message latency between the two processes.

3.2. Performance Portability

We first evaluate performance portability. In Figure 11, we provide the speedup obtained with the programs leveraging the libraries detailed in Section 3.1, with respect to the original single-core programs. The average speedup across all benchmarks is 2.4× for GPUs and 1.7× for multicores, respectively. Note that these results are obtained without modifying the program code, clearly illustrating that performance portability can be achieved using a library-based programming approach.

3.3. Overhead

In Figure 12, we measure the ratio of the execution time of the original program over the componentized program, and we report the corresponding slowdown for the various platforms. On average, the slowdown equals 0.99 for a single core, 0.95 for four cores, and 0.96 with a GPU, a fairly low performance penalty for portability. We also report the total overhead as the percentage of the original execution time of each platform in Figure 13. The average overhead for multicores and GPU equals 6.4% and 5.1%, respectively, with a respective maximum of 12.5% (JPEG) and 16% (BACKPROP) because of the small granularity of the tasks executed upon each call to the accelerator.

3.4. Overhead Breakdown

We further analyze the overhead in Figure 13. There are two main sources of overhead: the runtime and the componentization of the program. The overhead of componentization is a result of the additional code for encapsulating the task and some control code to communicate with the runtime system. The overhead of the runtime system includes the interprocess communication and accelerator selection. We distinguish between the two sources of overhead (runtime and componentization) by disabling (and bypassing) the runtime in one version; in that case, the sole source of overhead is the componentization. In Figure 13, the total overhead is broken down into the runtime and componentization overhead, as a percentage of the original execution time for each platform. On average, the componentization overhead equals 2.4% and 1.8% for the multicore and GPU, respectively, while the runtime overhead is about twice as high: 4% and 3.3%, respectively. The largest runtime overhead is incurred by BACKPROP on the GPU because of the task granularity (only one row is processed at a time

Table I. Description of Benchmarks

Benchmarks	Domains	Description of Benchmarks	Datasets	Algorithms	% time	Accelerators	Approx	Accuracy Metric
LEUKOCYTE	Medical Imaging	Detects and tracks rolling leukocytes in video microscopy of blood vessels	20 AVI 684 × 480 videos, number of frames for 5 to 30; 5 decomposed from Rodinia˜[Che et al. 2009], 15 more from Internet	CellDetection	5.4%	GPU		-
				EllipseTrack	94.6%	GPU		
NW	Bioinformatics	A global optimization method for DNA sequence alignment	20 pairs of matrices representing the pair of DNA sequences; each matrix is 16,384 × 16,384, randomly generated values	SequenceAlign	95.8%	GPU		-
BACKPROP	Pattern Recognition	Trains the weights of connecting nodes on a layered neural network	20,000 datasets; input layer size: 16,384–32,768; random input values and initial weights	Propagation	14.2%	Inexact CPU, CGRA, GPU, ML ASIC	√	$1 - MSE$
				WeightUpdate	45.7%	Inexact CPU, CGRA, GPU		
STREAM-CLUSTER	Data Mining	Online clustering	20,000 datasets; each contains 1,024–2,048 randomly generated 32–64-dimensional points	Dist	65.0%	Inexact CPU, CGRA, GPU	√	BCubed (B^3) [Amigo et al. 2009]
JPEG	Image Compression	Lossy compression for digital photography	20,000 copyright-free images; 100×100 (30KB) to 1,024 × 768 (2.3MB)	ColorConversion	10.9%	Inexact CPU, CGRA	√	SSIM [Wang et al. 2004]
				DCT	29.1%	Inexact CPU, CGRA, GPU		
				QT	18.9%	Inexact CPU, CGRA, GPU		
				HuffmanEncoding	37.1%	-		
KMEANS	Data Mining	Finds cluster centers that minimize the intraclass variance	20,000 datasets; each contains 4,096–12,288 randomly generated 32–96-dimensional points	Euclid_dist_2	38.1%	Inexact CPU, CGRA, GPU	√	BCubed (B^3)

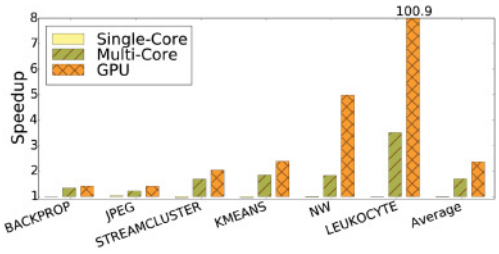

Fig. 11. Speedup over single-core execution.

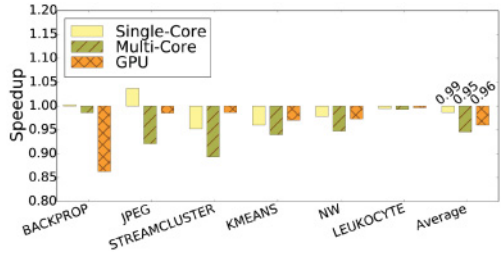

Fig. 12. Slowdown w.r.t. original programs.

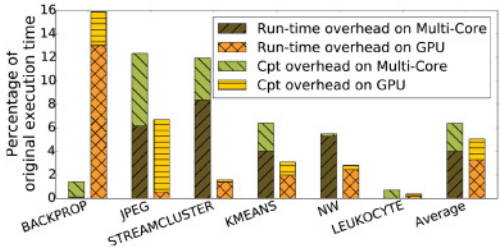

Fig. 13. Overhead.

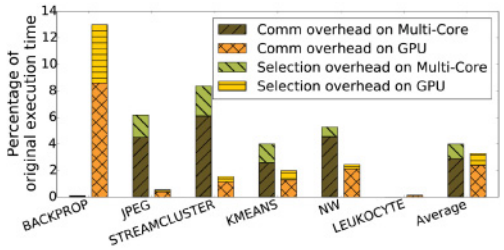

Fig. 14. Runtime overhead breakdown.

within the neural network), while the largest componentization overhead is incurred by STREAMCLUSTER and JPEG.

3.5. Runtime Overhead

We now analyze in more detail the overhead of the runtime system. We focus on the two main sources of runtime overhead: the communication overhead and the central runtime decision overhead. In Figure 14, we report the percentage of the total runtime overhead due to communication and selection, and we observe that the communication overhead accounts for about 75% of the runtime overhead.

While dynamic component loading is normally a costly operation, in part because of the interprocess communications with the central runtime, we have reduced its overhead by letting the local runtime cache the component code. The component code is Position Independent Code (PIC), linked with `-fPIC` and `-shared` flags. Because the actual component code address varies each time it is loaded, the performance can vary slightly for each component execution, because of, for example, instruction cache effects. The experiments highlight the importance of component code caching at the level of the local runtime: for programs with frequently called components, such as `JPEG`, a 0.62× slowdown becomes a 1.22× speedup.

4. RUNTIME EXPLORATION ON A SIMULATED SOC

We now take the proposed framework one step further and leverage it to drive approximate computing on a heterogeneous SoC. We consider a more realistic and more complex heterogeneous SoC architecture composed of a CPU along with a number of accelerators: a GPU, a CGRA (Coarse-Grain Reconfigurable Architecture), and an ASIC for machine-learning applications (see Figure 15).

We develop an *exploration strategy* within the runtime system that automatically selects the appropriate accelerator per algorithm. We leverage the fact that the same applications are usually run many times in most systems (embedded systems or data

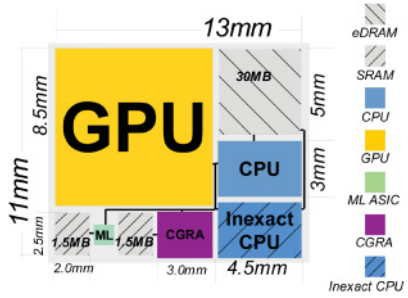

Fig. 15. SoC design.

Table II. Description of Accelerator Knobs

Accelerator	Description of Knobs	#Knobs	Value Range
CPU	*N* low-order bits of integer adders and multipliers	2	[0–31]
GPU	idem (in all streaming processors)	2	[0–31]
CGRA	idem (in all tiles)	2	[0–31]
ML ASIC	PLM multipliers	4	[0–3]

centers), so that the optimization can be performed across runs [Chen et al. 2012]. We implement an online simulated annealing strategy that collects information *across runs* and progressively tunes accelerator selection. This strategy, embedded within the runtime system, is application independent and low overhead enough to avoid overriding the performance/energy benefits of accelerators.

Approximate computing. Beyond architecture customization, *approximate computing* is emerging as the second most prominent avenue for energy reduction. Interestingly, approximate computing is highly compatible with the proposed approach because it also favors accelerators over cores, since the lack of error tolerance of control logic in cores limits the benefits of specialization, a problem accelerators can overcome [Esmaeilzadeh et al. 2012b]. As a result, we further augment the different accelerators with hardware knobs (see Figure 4) capable of varying the accuracy/energy tradeoff of each accelerator. Both the accelerators and the knobs are presented in more detail in Section 4.2.

In theory, approximate computing adds a level of software complexity for the programmer, for instance, in the form of language support [Sampson et al. 2011]. A more transparent method for the programmer has been proposed by Maggio et al. [2013] for managing the energy/accuracy tradeoffs in a homogeneous multicore: a global application error target is set, and control theory is used to steer accuracy/energy knobs. However, it requires an application-specific model of time and energy to drive the control, which would again translate into a significant programmer effort. Knob values can also be determined via offline training [Hoffmann et al. 2011], but the training has to be restarted for every accuracy target, and it is inherently dataset sensitive.

In our framework, we view accelerator knobs as parameters of the hardware platform just like the selection of the accelerator itself. The only consequence is that it expands the size of the solution space to be explored; the exploration strategy itself need not change. Consequently, we adapt the runtime to factor in these additional knob parameters, and in the process, we show not only that the runtime can tackle the two most important avenues for energy reduction (architecture specialization and approximate computing) but also that it can be easily retargeted to more complex cost functions.

4.1. Exploration Strategy

The exploration strategy is composed of two parts: (1) online simulated annealing to find the best combinations of accelerators and accelerator parameters and (2) a budget-based management of the exploration cost to ensure that the overhead of exploration does not outweigh the benefits of specialization and approximate computing.

4.1.1. Exploration via Online Simulated Annealing. The exploration is based on a simulated annealing (SA) algorithm. At every run of an application, the runtime performs

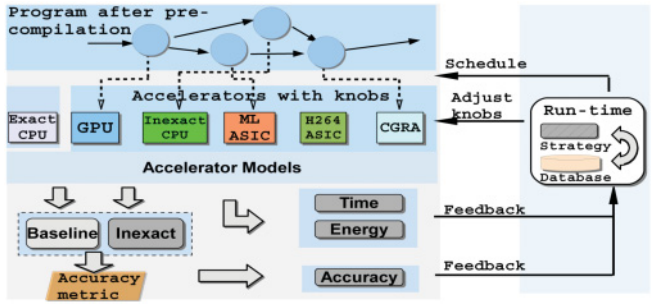

Fig. 16. Framework with accelerator accuracy knobs.

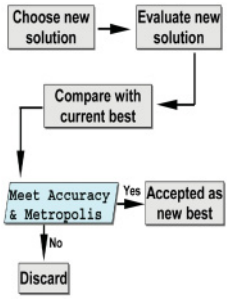

Fig. 17. One SA iteration.

one iteration of the simulated annealing algorithm. We recall that an SA iteration consists of choosing a new solution (here, an accelerator and a set of accelerator knob values), comparing that solution against the best solution so far (assessing its energy, performance, and accuracy), and either keeping that solution as the new best or discarding it (see Figure 17; the SA uses the classic Metropolis rule [Metropolis et al. 1953]). SA optimizes for performance and energy, respectively, as illustrated in Figure 18.

For each application, the runtime maintains a database of the results of previous executions for the across-run exploration (see Figure 16). Before the execution, it sets the accelerator and the accuracy/energy accelerator knobs based on the matching best results in the exploration so far, obtained during previous runs. For instance, for the JPEG application considered before, `CC`, `DCT`, and `QT` can all tolerate inexact computations, and we assume they are executed on an inexact accelerator with two knobs: `knob_adder` and `knob_multiplier`, and then the runtime obtains the following vector from the database: `<cc_knob_adder, cc_knob_multiplier, dct_knob_adder, dct_knob_multiplier, qt_knob_adder, qt_knob_multiplier>`. After the execution, the runtime records the accelerator knob values, energy, performance, and achieved application accuracy.

The goal of the exploration is to find the accelerator and the accelerator knob values that satisfy the different objectives. There are usually three main types of objectives: energy, execution time (performance), and error. The runtime treats energy and execution time differently from error: the weighted sum of energy and execution time will be minimized, but a solution for which the estimated error is below the target error will not be accepted; note that the programmer can provide the respective weights of energy and execution time, as well as the error target, as we will describe in Section 4.1.3.

4.1.2. Budget-Based Exploration. The main challenges of the SA-driven exploration are the following: (1) perform the exploration across production runs (i.e., different datasets), (2) minimize the energy and time overhead of the exploration, and (3) adjust to changing error targets set by the programmer/user.

The first two points are particularly challenging: either we compare execution time/energy between two runs with two different datasets and the comparison is likely to be meaningless, or we run the same dataset again using a default (initial) accelerator and set of knob values to obtain comparable absolute execution time/energy improvements across runs but the overhead is higher than the benefits, so the approach is similarly pointless. Leveraging techniques pioneered for iterative optimization in data centers [Chen et al. 2012], where comparing the impact of compiler optimizations across runs with distinct datasets raises similar challenges, we introduce energy and execution time *budgets* and we manage it as follows *across runs* of the same application.

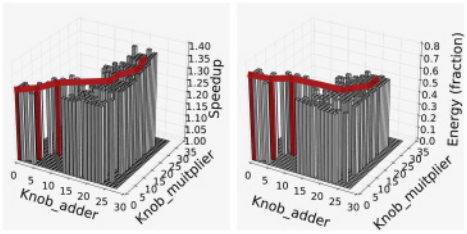

Fig. 18. SA walks for performance (left) and energy (right). The graphs assume an approximate CPU running the streamcluster benchmark.

Fig. 19. Energy/time profile during accelerator exploration.

Every time we want to try a new solution during exploration (i.e., a new accelerator and set of knob values), we also run the same dataset using the default accelerator and set of knob values to obtain an absolute time/energy improvement with respect to that baseline; in other words, there is a *redundant* execution of the same dataset. We consider this redundant run both an energy and execution time *expense* (or overhead). However, every time a new, better solution is found, we estimate the *benefit* (delta) in energy and time, and we assume that every subsequent run will bring a comparable benefit (more precisely, a benefit weighted by the execution time ratio of the two datasets), because, hypothetically, each run would have incurred higher energy and execution time with the previously selected accelerator and set of knobs. The runtime accumulates these benefits until they can be *spent* again during another SA exploration. We keep the overhead cost low by only triggering exploration along with the redundant (baseline) run using the default selection when the execution time of the run is lower than the average execution time of all runs seen so far.

This budget-based approach only works well when a given application is run many (thousands of) times on the same system. But, almost by definition, this is the case of the most popular/important applications, whether it is in data centers or in embedded systems. On the very first run of the application, we set an initial budget with the (likely) assumption that a better solution will be found after a number of iterations. If no better solution is found, the overhead of this exploration averages out over time, as the application keeps running (see also Figure 19).

4.1.3. Additional Support for Approximate Computing. With respect to approximate computing, we introduce two software APIs, @APP_ACCURACY_BOUND and @APP_ACCURACY_METRIC (see lines 3–5 in Figure 4), that enable the programmer to provide an accuracy lower bound and a method for estimating the error. The programmer can also provide the desired weight for energy and execution time via @APP_ENERGY_WEIGHT and @APP_SPEEDUP_WEIGHT (see line 5 in Figure 4). Moreover, we also provide a hardware API to the runtime for tweaking the accelerator knobs and their value ranges (see Figure 21). We show an example application of approximate computing in Figure 20 for a JPEG running on an inexact core with two knobs controlling the adder and multiplier bit widths: the top picture shows the result when the 28 and 27 lower-order bits are masked for the adder and multiplier, respectively; nine and seven lower-order bits are masked in the bottom picture for the adder and multiplier, respectively.

4.2. Methodology

SoC simulator. We developed an SoC simulator to evaluate the proposed framework, modeling one general-purpose CPU, one CPU with approximate operators, and three

(a) Knob=(28,27), SSIM=0.42

(b) Knob=(9,7), SSIM=0.93

Fig. 20. The impact of hardware knobs on JPEG.

```
1  struct Knob {
2      string knob_name;
3      KnobRange range;
4      KnobValue currentValue;
5      void set_value(KnobValue v);
6      ...
7  };
8  void InitKnob (Knob k, KnobValue v=default);
9  void AdjustKnob (Knob k, KnobValue delta_v);
10
11 // Implement set_value function for CGRAKnobAdder
12 void set_value_cgraAdder (KnobValue v)
13 {/* Tailor v bits computation of adder in CGRA */}
14
15 // Create one knob for CGRA
16 Knob CGRAKnobAdder("cgraAdder", Range(0.4,1,0.1),
17                    ...,  set_value_cgraAdder);
```

Fig. 21. Hardware API.

accelerators: a GPU, a CGRA, and a machine-learning (ML) ASIC (see Figure 15). We synthesized, placed, and routed the accelerators and used estimated die areas for the CPUs, assuming a 32nm chip technology. On-chip memory is organized as follows: the two CPUs share a 30MB on-chip eDRAM [Wang et al. 2009], and the ML ASIC and CGRA have their own private 1.5MB SRAM bank. The accelerators and CPUs are connected by a 2D mesh-based on-chip network. We use the BookSim 2.0 [Jiang et al. 2013] NoC simulator in the experiments, with the following configuration: 1.5GHz, 128-bit link width, eight virtual channels per router, and five buffers in each channel; we use wormhole packet switching with 32 flits per packet.

For the CGRA, ML ASIC, and approximate operators used in both the CPU and the GPU, we have synthesized the Verilog circuits using the Synopsys Design Compiler and the TSMC 32nm library GP (General Purpose), HVT (High Voltage Threshold); placement and routing were performed using Synopsys ICC.

Here, we present the different architectures/accelerators used in this study and the source of approximation: a CPU (and its exact version used as a baseline), a GPU, a CGRA, and two ASICs. (The different accelerator knobs are also summarized in Table II.)

General-purpose and inexact CPU. The CPU is an Intel Atom N series processor. We have applied approximate computing at the level of the ALUs. We measure power consumption for the exact CPU using hardware counters. For the inexact CPU, we have implemented in Verilog, then obtained the postlayout energy of $(32 - N)$-bit integer adders and $(32 - N)$-bit integer multipliers, with N low-order bits removed, a standard way for modulating both the error and dynamic power consumed by an operator [Lingamneni et al. 2011]. The adders/multipliers are also used for fixed-point instead of floating-point operations.

GPU. As for the GPU, we only introduce approximation at the level of the ALUs, albeit in all streaming processors (SMs) of an embedded NVidia Tegra GPU. We use the power model proposed by Hong and Kim [2010], and for each algorithm, we obtain the number of active SMs from the compilation phase.

Approximate CGRA. We use the elastic CGRA proposed by Huang et al. [2013]. It is a grid of word-sized tiles with one ALU each. We again introduce the aforementioned approximate adders and multipliers in these tiles. We use their tool chain to compile,

place, and route the code on the CGRA, and we deduce the number of active tiles for each algorithm.

Machine-learning ASIC. We implement in Verilog, synthesize, and place and route a Multi-Layer Perceptron (90 inputs, 16 hidden neurons, 10 outputs), slightly larger than the one proposed by Temam [2012], which was shown to be able to tackle 90% of the tasks provided in the UCI Machine-Learning repository. We use probabilistic logic minimization (PLM) to create multipliers that barely affect the MLP accuracy [Lingamneni et al. 2011] and obtain four different configurations.

Benchmarks and datasets. Because `LEUKOCYTE` and `NW` do not lend well to approximate computing, in this section, we only consider the last four benchmarks of Table I (the ones marked with a check mark in the *Approx* column). For each benchmark, we collected 20,000 distinct datasets; each dataset is run only once in order to realistically assess the impact of the runtime system in production runs.

The error metric for each application is also indicated in Table I. For `JPEG`, we use the *Structural Similarity* (SSIM) metric. For `KMEANS` and `STREAMCLUSTER`, we use the BCubed (B^3) clustering quality metric [Amigó et al. 2009], which computes the homogeneity and completeness of clustering. Finally, for `BACKPROP`, we simply use the average output error.

Time measurements. We have implemented a timed transaction-level simulator of the SoC; the runtime system is run on the exact CPU. The Timed Transaction-Level Modeling (TTLM) [Lu et al. 2012] is a popular modeling approach in system-on-chip design. TTLM is based on the notion that an architecture is composed of many independent IP blocks linked via an interconnect. The principle of TTLM is to model the functional behavior of each IP block implemented as a separate module in the simulator, and to add timing annotations to each outgoing request of an IP block. The motivation for this approach is that many IP blocks (e.g., FPGAs/CGRAs, ASICs) have a very predictable time behavior, which does not warrant the typical cycle-level/bit-level simulators used in the micro-architecture domain such as gem5 [Binkert et al. 2011]. We briefly explain how we built the time annotations for each accelerator. For the CGRA, we have precisely estimated the number of cycles required for executing each algorithm mapped to the accelerator, as well as the cycle time. For the ML ASIC, the time required to process each input is constant, and thus, it was again precisely evaluated. For ML ASIC, we emulate a DMA and we consider the time required to fetch a DMA buffer row to be constant. For the GPU and CPU, we use the execution times collected on the NVidia and Intel platforms, respectively, for each algorithm instance on each dataset. We use BookSim 2.0 with the application traces obtained using Intel Pin [Luk et al. 2005] to simulate the NoC behavior.

Energy measurements. The SoC energy is the sum of the energy of the accelerators, the NoC, and the memories (SRAMs and eDRAM). For the NoC energy, we use Orion2.0 [Kahng et al. 2009], using the traffic load obtained with BookSim 2.0. For the SRAMs and eDRAM, we use CACTI 5.3 [Cacti 5.3 2008] and electrical parameters from IBM eDRAM [Wang et al. 2009], respectively. For the CPUs and accelerators, we use standard CAD tools to derive so-called average power models for operators or whole accelerators. We obtain the average power P_{avg} for each accelerator/operator; the average energy E_{avg} of a run of duration T is calculated as $E_{avg} = P_{avg} \times T$.

4.3. Performance Evaluation

We have run all benchmarks on all their 20,000 datasets using the exploration strategy. The purpose of the exploration strategy is to select both the best accelerator and the

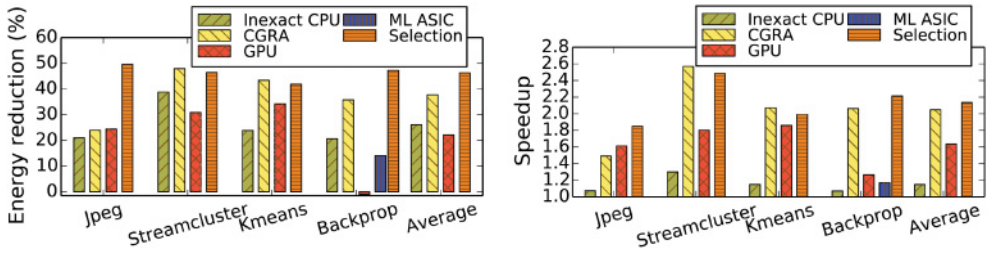

Fig. 22. Total energy reduction (left) and speedup (right) with accelerator selection, after 20,000 runs; the exact CPU is our baseline.

best accelerator configuration (knobs). In Figure 22, we report the aggregated results for an error target of 5% for all the benchmarks; both energy reduction and speedups are provided with respect to the execution on the exact core.

In each benchmark cluster, the bars correspond to the performance obtained with the standard runtime system (fixed accelerator, only accelerator knobs are varied), and the rightmost bar *Selection* in each cluster corresponds to the full exploration. For all benchmarks, the exploration strategy can successfully select the accelerator that provides the best performance and adjust its knobs. For instance, for `JPEG`, the `CGRA` is selected for `CC`, and the `GPU` for `DCT` and `QT`. For `BACKPROP`, the `ML ASIC` is selected for `Propagation`, and the `CGRA` for `WeightUpdate`. We observe that the *Selection* bar is slightly lower than the best fixed-accelerator performance for `STREAMCLUSTER` and `KMEANS` because, for both of these applications, the CGRA is almost consistently better energy-wise and performance-wise compared to the other accelerators; therefore, when the SA exploration decides to try out another accelerator, it can only degrade performance, even if it quickly recovers from that mistake. However, as explained earlier, the relative merit of each accelerator can change with the accuracy target, so the automatic selection of accelerators is ultimately a safer strategy, even if it does not always yield the best improvements. Note that the energy reduction for the ML ASIC is lower than that of the CGRA, even though the ASIC is a significantly more energy-efficient architecture; this is due to the fact that only the forward path of the neural network is implemented in the ASIC, while `BACKPROP` performs both the forward and backward path.

Overall, the exploration strategy can achieve 46.2% energy reduction on average, and up to 49.6%, and a speedup of 2.1× on average, and up to 2.48×. Recall that all the programmer needs to provide to achieve these results are properly annotated accelerator versions of some algorithms, the error method, and the error target value.

In Figure 23, we report the cumulated energy reduction and speedup profiles (solid lines). We also show how the accuracy ($1-error$) evolves across runs, and the horizontal red solid line marks the target accuracy (set at 95%, i.e., a 5% error target for all error metrics). For all benchmarks, in spite of a slight degradation at the beginning, corresponding to the initial budget invested in exploration, the budget-based exploration successfully converges to significant speedups and energy reductions while meeting the error target.

We also separately measured the exploration overhead (baseline run, poor configurations, SA) and report the performance without overhead with dashed lines in the same figures. While the overhead degrades both the energy improvement and speedup, the loss remains small in all cases; that is, the runtime efficiently manages the cost of the exploration. We also report the accuracy evolution (green line). It is gradually approaching but always above the accuracy bound line (the red line) as the exploration progresses. Still, in order to avoid any accuracy issue during production runs,

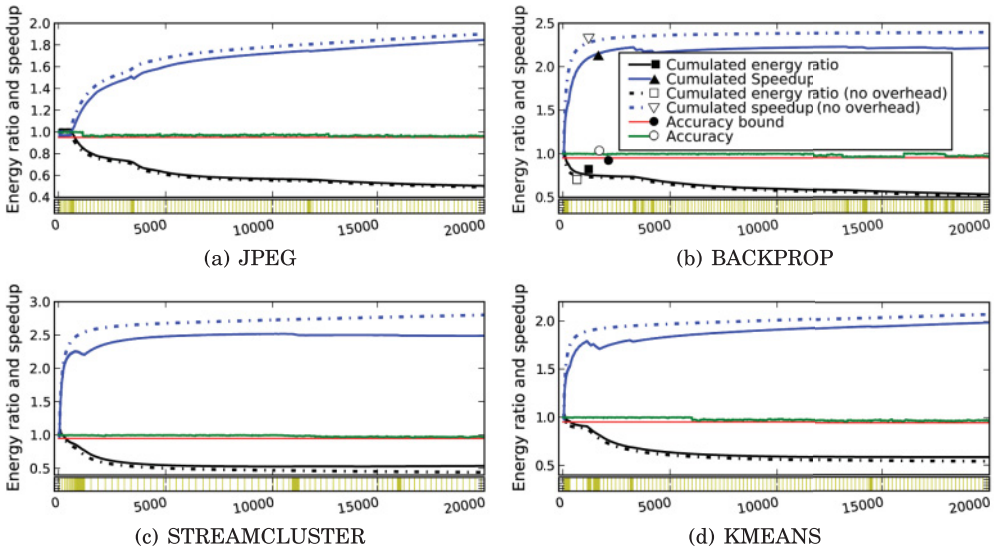

Fig. 23. Cumulated energy reduction and speedup for accelerator selection. All figures share the same legend, and markers are used to distinguish curves.

a simple alternative is to perform a number of SA iterations offline on training data until the error target is met, and to repeat this for all possible error targets by steps of 5%. The actual production runs are then started with the current state of the SA exploration.

The vertical ticks at the bottom of each graph in Figure 23 indicate when a new configuration is tried out. Initially, the budget-based exploration algorithm more aggressively explores accelerator options and knobs, and once well-performing configurations are found, the algorithm becomes less aggressive, and fewer explorations are performed. Although the total exploration space is quite large (the product of the number of accelerators and their knob values), typically only 5% of the space is effectively evaluated, which suggests that the exploration algorithm is converging effectively.

5. RELATED WORK

We distinguish the related work on programming for heterogeneity, algorithm-based programming, and approximate computing.

5.1. Programming for Heterogeneity

5.1.1. Languages. We advocate the generalized usage of libraries to expose accelerators to programmers. In contrast, many research works propose either different programming models or domain-specific languages. The programming models of *CUDA* and *OpenCL* give more complete control to programmers, but they also require them to understand and directly deal with low-level concepts of accelerators. Brown et al. [2011] propose to use Domain-Specific Languages (DSLs) to map high-level application code onto heterogeneous systems using *Delite*. *Chapel* [Sidelnik et al. 2012] is an object-oriented parallel programming model that is more focused on data parallelism on GPU and multicores. In the programing models of HMPP [Dolbeau et al. 2007], *SEJITS* [Catanzaro et al. 2009], *OpenMPC* [Lee and Eigenmann 2010], *OMPSS* [OMPSs 2010], and *OpenACC* [OpenACC 2013; Grewe et al. 2013], the syntactic transformations are less significant, though there is still an underlying programming model: annotations or pragmas are associated with sections of sequential code to

modify or restrict their semantics so that compilers can automatically transform them into parallel or accelerator code. Finally, a number of programming languages focus on streaming applications, such as PeakStream [Papakipos 2006], which relies on a special implementation of data structures (arrays). *Lime* [Dubach et al. 2012] extends Java to directly support pipeline-based task parallelism and data parallelism.

5.1.2. Environments. One of the earliest related works on programming environments is *Charm++* [Kalé and Krishnan 1993]. While not designed for heterogeneous multicores, it can tackle distributed-memory multicores, and in that respect it relates to accelerators that can each have their own address space. More recently, *Sequoia* [Fatahalian et al. 2006] is especially focused on explicitly introducing the memory hierarchy into the programming model and on divide-and-conquer parallelization, albeit using self-contained and isolated units instead of functions. *Merge* [Linderman et al. 2008] is focused on mapping programs with a high degree of parallelism on heterogeneous systems. But it exclusively relies on a Map-Reduce environment for facilitating the task of writing and managing highly parallel programs.

Harmony [Diamos 2008] introduces an environment (especially a runtime) where a program is decomposed into compute kernels. The main emphasis of *Harmony* is on the ability to find and manage dependences among the compute kernels dynamically, in the same spirit as between instructions of an out-of-order processor, in order to optimize their scheduling. *PEPPHER* [Kessler et al. 2012] and *StarPU* [Augonnet et al. 2009] focus on supporting numerical applications with abundant fine-grained data parallelism. However, memory management still requires explicit data partitioning by the programmer. Moreover, the task granularity preferred by *StarPU* is small, and it has so far been applied to simple numerical kernels, such as LU decomposition. In both cases, we make a different tradeoff: we voluntarily expose only a simple sequential machine model to the programmer, so components execute in sequence, but parallelism can occur *within* components.

Runnemede [Carter et al. 2013] is an HW/SW codesign framework for achieving low energy in large-scale systems. However, it focuses on the extreme parallelism of fine-grained tasks in order to leverage the large number of relatively simple cores used in the *Runnemede* architecture. *Reflex* [Lin et al. 2012] aims at making low-power processors of mobile heterogeneous multicore systems more accessible to developers. The runtime of Reflex is distributed among the different processors, built upon the shared address space. We do not assume a shared address space among our accelerators, though we can naturally deal with one.

Qilin [Luk et al. 2009] shares several features with *PeakStream* [Papakipos 2006], especially its programming model and runtime design, except that it maintains a database of all previously executed programs to capture the relationships between program, input, architecture characteristics, and execution time in order to improve load balancing.

MATCH [Nayak et al. 2000] also uses library wrappers to encapsulate MATLAB libraries on DSP, FPGA, and PowerPC platforms. Unlike our runtime, *MATCH* cannot dynamically schedule these wrappers among the different accelerators. Finally, *MAGMA* [MAGMA 2013] is a linear algebra library that can leverage multicores and GPUs. They manually partition the algorithms into CPU and GPU parts using domain-specific knowledge.

5.2. Algorithm-Based Programming

With respect to algorithm substitution, Ansel et al. [2009] highlight the notion of algorithm-centric programming with *PetaBricks*, providing runtime support for dynamically selecting the best algorithm version. Phothilimthana et al. [2013] extend

PetaBricks to support GPUs by generating code for these architectures; our runtime is based on leveraging existing libraries instead of code generation, and it is not specific to GPUs. Li et al. [2012] focus on methods for organizing algorithm information so that substitution can be managed automatically by a precompiler. Becchi et al. [2010] propose a method for allowing legacy binaries to run on heterogeneous platforms by intercepting function calls and redirecting the control to accelerated versions of the library (if available). The identification of such libraries is ad hoc; there is no notion of systematically wrapping legacy code with wrappers.

The similar approach (called component-based design) is well studied in the HW/SW codesign area [Cesário et al. 2002; Arató et al. 2005; Cheung et al. 2007]. The hardware IPs are wrapped as components in order to lift the work abstraction for designers, and the partitionable component is introduced to unify the interface of hardware and software components during the copartitioning phase. Martin et al. [2001] and Roop et al. [2000] address the problem of automating the component selection during the design phase of the embedded system. Communication issues between components are studied in Rincon et al. [2007] and Sgroi et al. [2001]. In more general terms, HW/SW codesign leverages component-based design to rapidly prototype and functionally simulate complex hardware/software systems. We instead apply similar notions to the programming and runtime management of heterogeneous SoCs.

5.3. Approximate Computing

Several studies have shown that applications such as machine learning, multimedia, and computer vision, among others, can tolerate hardware or software errors [Kruijf et al. 2010]. As a result, several research groups have investigated tradeoffs between accuracy and performance [Agarwal et al. 2009], robustness [Rinard 2006], fault tolerance [Chakrapani et al. 2008], and energy consumption [Sampson et al. 2011; Ansel et al. 2011]. In this study, we factor in the current architecture trend toward heterogeneous accelerators [Esmaeilzadeh et al. 2011], and we focus on approximate accelerators. Approximation is introduced at the level of hardware (operators or memory) but is managed by software.

There are different approaches for introducing the notion of approximate computing within programs. Sampson et al. [2011] propose to manage approximate computing at the programming language level and introduce approximate type qualifiers. Techniques such as loop perforation [Misailovic et al. 2010], code perforation [Agarwal et al. 2009], and task skipping [Rinard 2006] let the compiler automatically transform exact codes into approximate versions. Such approximate codes skip some noncritical computations (e.g., loop iterations or tasks).

Several frameworks have been proposed to investigate tradeoffs between the accuracy, execution time, and energy, such as *Green* [Baek and Chilimbi 2010], *Eon* [Sorber et al. 2007], *PowerDial* [Hoffmann et al. 2011], and *PetaBricks* [Ansel et al. 2011]. *Green* explores approximate implementations using perforation techniques; its training is conducted offline with online recalibration. *Eon* is more focused on energy saving for mobile devices by adjusting the duty cycle of a GPS receiver. *PowerDial* is a framework that tunes program parameters, so they can adapt to the modified underlying hardware. Ansel et al. [2011] extend *PetaBricks* to automatically tune the algorithmic parameters that are critical to execution time and accuracy. We instead focus more on performance portability via accelerator selection and hardware parameter tuning.

Other research groups have specifically focused on hardware techniques to trade off accuracy for energy [Liu et al. 2009; Alvarez et al. 2005; Esmaeilzadeh et al. 2012a, 2012b]. These techniques aim at designing energy-efficient architectures, which is complementary with our approach.

6. CONCLUSIONS

We propose a framework to improve the programmability of complex heterogeneous SoC architectures by nonexpert programmers while achieving performance portability. The framework relies on the notion of library-based or algorithm-centric programming. By providing simple semantic information on the program algorithms, and by wrapping existing libraries and/or custom algorithm implementations, programmers can develop programs that are portable across many accelerators without modifying a single line of (glue) code. Moreover, the semantic information empowers the runtime to automatically select the most appropriate accelerator (and its settings) for each algorithm. For that purpose, we develop an exploration strategy, which carefully manages the overhead of trial runs to maximize benefits during program execution.

We demonstrate the performance portability of our approach by componentizing/rewriting six benchmarks; on a real platform composed of a multicore and a GPU, we achieve a speedup of 2.4× on the GPU and 1.7× on the multi-core, on average, over single-core execution. We also show that the overhead of wrapper code and runtime management is low at 5.1% for GPUs and 6.4% for multicores. Finally, we apply the approach to a simulated SoC with several accelerators, capable of approximate computing, and we show that the exploration strategy can achieve an energy reduction of 46.2% and a speedup of 2.1× on average while still meeting the desired application error targets.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. Y. Fang and H. Li are supported by the National Natural Science Foundation of China (NSFC) under grants No. 61176040 and 61221062. Yunji Chen is supported by the NSFC under grants No. 61222204, 61221062, and 61303158, the Strategic Priority Research Program of CAS under grant No. XDA06010403, and the China 10000-talents program. S. Fang, Z. Du, Y. Huang, and C. Wu are supported by the NSFC under grants No. 60873057, 60921002, 60925009, 61033009, and 61202055; the National High Technology Research and Development Program of China under grant No. 2012AA010902; and the National Basic Research Program of China under grant No. 2011CB302504. L. Eeckhout is supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement No. 259295. O. Temam is supported by a Google Faculty Research Award, the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), and the China 1000-talents program.

REFERENCES

A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. 2009. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures*. Technical Report. MIT.

C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computing* 54, 7 (2005), 922–927.

E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo. 2009. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information Retrieval* 12, 4 (2009), 461–486.

J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 38–49.

J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 85–96.

P. Arató, Z. A. Mann, and A. Orbán. 2005. Extending component-based design with hardware components. *Science of Computer Programming* 56, 1–2 (2005), 23–39.

C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. 2009. StarPU: A unified platform for task Scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference (Euro-Par)*. 863–874.

W. Baek and T. M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 198–209.

M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar. 2010. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 82–91.

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 89–100.

Cacti 5.3 . 2008. CACTI. Available at: http://hpl.hp.com/research/cacti/.

N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. 2013. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 198–209.

B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. 2009. SEJITS: Getting productivity and performance with selective embedded JIT specialization. *Programming Models for Emerging Architectures* 1, 1 (2009), 1–9.

W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, and M. Diaz-Nava. 2002. Component-based design approach for multicore SoCs. In *Proceedings of the 39th Annual Design Automation Conference (DAC)*. 789–794.

L. N. B. Chakrapani, K. K. Muntimadugu, A. Lingamneni, J. George, and K. V. Palem. 2008. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 187–196.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

Y. Chen, S. Fang, L. Eeckhout, O. Temam, and C. Wu. 2012. Iterative optimization for the data center. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 49–60.

P. H. Cheung, K. Hao, and F. Xie. 2007. Component-based hardware/software co-simulation. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*. 265–270.

G. Diamos. 2008. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 197–200.

R. Dolbeau, S. Bihan, and F. Bodin. 2007. HMPP: A hybrid multi-core parallel programming environment. In *Proceedings of the Workshop on GPGPU*. CAPS Entreprise, 1–5.

C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. 2012. Compiling a high-level language for GPUs (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1–12.

H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. 365–376.

H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012a. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 301–312.

H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012b. Neural Acceleration for General-Purpose Approximate Programs. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–6.

K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. 2006. Sequoia: programming the memory hierarchy. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. Article 83.

D. Grewe, Z. Wang, and M. F. P. O'Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10.

G. Heineman and W. Councill. 2001. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman, Boston, MA.

H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 199–212.

S. Hong and H. Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. 280–289.

Y. Huang, P. Ienne, O. Temam, and C. Wu. 2013. Elastic CGRAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*. 171–180.

Intel. 2011. *Intel64 and IA-32 Architectures Software Developer's Manual*. Intel.

N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 86–96.

A. B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 423–428.

L. V. Kalé and S. Krishnan. 1993. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 91–108.

C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. L. Traff, and S. Pllana. 2012. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*. 1403–1408.

M. D. Kruijf, S. Nomura, and K. Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. 497–508.

T. Lane and the Independent JPEG Group. 1991. Libjpeg. Available at: http://libjpeg.sourceforge.net/.

S. Lee and R. Eigenmann. 2010. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

H. Li, W. He, Y. Chen, L. Eeckhout, O. Temam, and C. Wu. 2012. SWAP: Parallelization through algorithm substitution. *IEEE Micro* 32, 4 (2012), 54–67.

F. Lin, Z. Wang, and R. LiKamWa. 2012. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 13–24.

M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. 2008. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 287–296.

A. Lingamneni, C. Enz, J. L. Nagel, K. Palem, and C. Piguet. 2011. Energy parsimonious circuit design through probabilistic pruning. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 1–6.

S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. 2009. *Flicker: Saving Refresh-Power in Mobile Devices through Critical Data Partitioning*. Technical Report. Microsoft Research.

K. Lu, D. Muller-Gritschneder, and U. Schlichtmann. 2012. Accurately timed transaction level models for virtual prototyping at high abstraction level. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 135–140.

C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

C. K. Luk, S. Hong, and H. Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 45–55.

M. Maggio, H. Hoffmann, M. Santambrogio, A. Agarwal, and A. Leva. 2013. Power optimization in embedded systems via feedback control of resource allocation. *IEEE Transactions on Control Systems Technology* 21 (2013), 239–246.

MAGMA 2011-2013. MAGMA: http://icl.cs.utk.edu/magma/index.html. (2011–2013).

G. Martin, R. Seepold, T. Zhang, L. Benini, and G. De Micheli. 2001. Component selection and matching for IP-based design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 40–46.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. 2010. Quality of Service Profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 25–34.

A. Nayak, M. Haldar, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. 2000. A library based compiler to execute MATLAB programs on a heterogeneous platform. In *Proceedings of the Conference on Parallel and Distributed Computing Systems (PDCS)*. 1–9.

J. Oberleitner and T. Gschwind. 2002. Composing Distributed Components with the Component Workbench. In *Proceedings of the 3rd International Conference on Software Engineering and Middleware (SEM)*. 102–114.

OMPSs 2010. OMPSs. https://pm.bsc.es/ompss. (2010).

OpenACC 2013. OpenACC 2.0. Available at http://www.openacc-standard.org/.

M. Papakipos. 2006. The PeakStream platform: High-productivity software development for multi-core processors. In *Proceedings of the Los Alamos Computer Science Institute, Workshop on Heterogeneous Computing (LACSI)*. 1–10.

P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. 2013. Portable performance on heterogeneous architectures. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 431–444.

M. Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS)*. 324–334.

F. Rincon, J. Barba, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. Lopez. 2007. Unified Inter-Communication Architecture for Systems-on-Chip. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP)*. 17–26.

P. S. Roop, A. Sowmya, and S. Ramesh. 2000. Automatic component matching using forced simulation. In *Proceedings of the 13th International Conference on VLSI Design (VLSI Design)*. 64–69.

A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 164–174.

M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. 2001. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*. 667–672.

A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. Padua. 2012. Performance portability with the Chapel language. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*. 582–594.

J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. 2007. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys)*. 161–174.

O. Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. In *Proceedings of the 39th annual international symposium on Computer architecture (ISCA)*. 356–367.

G. Wang, D. Anand, N. Butt, A. Cestero, M. Chudzik, J. Ervin, S. Fang, G. Freeman, H. Ho, B. Khan, B. Kim, W. Kong, R. Krishnan, S. Krishnan, O. Kwon, J. Liu, K. McStay, E. Nelson, K. Nummy, P. Parries, J. Sim, R. Takalkar, A. Tessier, R. M. Todi, R. Malik, S. Stiffler, and S. S. Iyer. 2009. Scaling deep trench based eDRAM on SOI to 32nm and Beyond. In *Proceedings of the IEEE International Electron Devices Meeting (IEDM)*. 1–4.

Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing.* 13 (2004), 600–612.

Received June 2013; revised December 2013; accepted January 2014