DOSAGE: Generating Domain-Specific Accelerators for Resource-Constrained Computing

Ankur Limaye
Department of Electrical & Computer Engineering
University of Arizona
Tucson, USA
ankurlimaye@email.arizona.edu

Tosiron Adegbija
Department of Electrical & Computer Engineering
University of Arizona
Tucson, USA
tosiron@arizona.edu

***Abstract*—Integrating low-overhead domain-specific accelerators with low-energy general-purpose processors can improve the processors' performance efficiency in resource-constrained systems (e.g., embedded systems). However, current function-based approaches for designing domain-specific accelerators require substantial programmer efforts for hardware/software partitioning and program modifications to access the best available hardware accelerators. This paper presents *DOSAGE*, an LLVM compiler-based methodology to generate domain-specific accelerators for resource-constrained computing systems. Given a set of applications, DOSAGE automatically identifies and ranks the recurrent and similar code blocks that would benefit the most from hardware acceleration, based on the code blocks' composition. We illustrate the benefits of the proposed approach using a case study that involves generating domain-specific accelerators for a diverse set of healthcare applications and evaluate the accelerators via FPGA-based prototyping. Compared to a base low-resource RISC-V processor, DOSAGE accelerators improved the system's performance and energy by 24.85% and 8.54%, respectively. Furthermore, compared to a state-of-the-art function-based accelerator generation approach, DOSAGE eliminated the function-level granularity constraint of the generation process and reduced the number of required accelerators—and, in effect, the interfacing overhead—by 33.33%, while achieving equal or better program coverage and performance/energy results.**

***Index Terms*—Domain-specific accelerators, hardware software co-design, low-energy, embedded systems, resource-constrained computing**

I. Introduction

The current generation of microprocessors has hit performance and power walls due to the slowdown in technology scaling and cooling constraints for increasing power density. Current microprocessors' inability to sustain the historically observed performance scaling has necessitated design alternatives for improving the execution efficiency of emerging systems. One design approach is an entirely hardware-based solution: *using custom-designed ASIC modules* for the target applications. An alternative approach involves hardware/software co-design, wherein the applications (entirely or selective code blocks) are executed on *hardware accelerators* attached to general-purpose microprocessors.

Fig. 1 illustrates the trade-offs between programming flexibility and performance efficiency offered by these two approaches. Custom-designed ASIC modules are typically generated using the high-level synthesis (HLS) flow. In the HLS flow, HLS tools (e.g., Vivado HLS [1], Microchip's LegUp [2]) compile applications written in a high-level programming language (e.g., C/C++) to synthesizable modules in hardware description languages (HDL) (e.g., Verilog/VHDL). The HLS flow is highly automated, and the resulting custom ASIC designs are typically optimal for specific applications. However, ASICs exhibit limited flexibility and programmability since they are fabricated for specific applications.

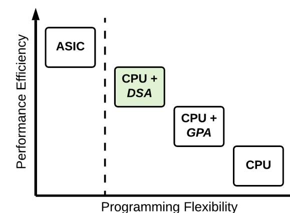


Fig. 1. Programming flexibility and performance efficiency trade-offs for ASIC designs and three CPU-based systems: only CPU, with general-purpose accelerators (GPA), and with domain-specific accelerators (DSA)

Alternatively, utilizing hardware/software co-design to integrate hardware accelerators with microprocessors provides much more flexibility than ASIC modules [3], [4]. These hardware accelerators execute select program code blocks more efficiently than the microprocessor pipeline. However, the applications may need to be redesigned to optimally access these accelerators, or the accelerators may need to be redesigned if the application changes. These accelerators can be classified into two categories: *general-purpose* and *domain-specific* accelerators.

General-purpose accelerators (e.g., GPUs, DSPs, Audio/Video codecs) [5], [6] are commonly found on microprocessor systems. These accelerators have standardized application program interfaces (APIs), making it easier to port/redesign the application to access them, hence offering long-term stability. However, these accelerators typically have a high area footprint that may be prohibitive and underutilized in resource-constrained computing systems.

Domain-specific accelerators, on the other hand, are customized for common code blocks within an application domain [7], [8]. They are more optimized—they have better performance and smaller area footprint—than general-purpose accelerators for the specific application domain. However,

978-1-6654-3922-0/21/$31.00 ©2021 IEEE

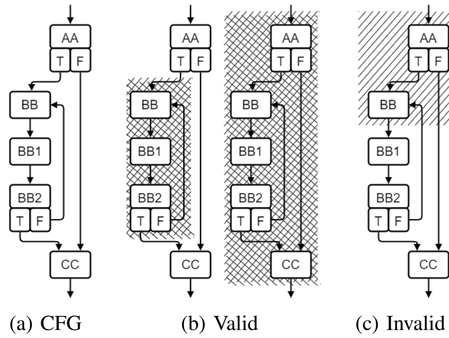


Fig. 2. *Super-block* illustration: (a) Example CFG, (b) Valid *super-blocks*, (c) Invalid *super-block*

these benefits come at the cost of increased hardware/software co-design effort and more application code modifications to interface with the accelerators since the APIs may not always be stable. In addition, current design approaches generate accelerators that may not be reusable across different applications within an application domain [9]. Our work aims to address these drawbacks posed by current approaches.

In this paper, we present ***DOSAGE***, a programmer-agnostic methodology for domain-specific accelerator generation for resource-constrained systems. DOSAGE is an LLVM compiler-based semi-automated approach to minimize programmer efforts for code modifications to interface with the accelerators. Given a set of applications, the DOSAGE workflow first identifies the computationally similar code blocks among the applications that would highly benefit from acceleration. Thereafter, DOSAGE applies a ranking heuristic that uses the code blocks' size and frequency to choose the best candidates—the appropriate *code block dosage*, so to speak—for hardware acceleration. To evaluate DOSAGE, we used a case study of a set of diverse real-world healthcare applications from the HERMIT benchmark suite [10]. Using the DOSAGE methodology, we generated a set of domain-specific accelerators for a resource-constrained system running the HERMIT workloads, and evaluated the generated accelerators via FPGA-based prototyping. Compared to a base RISC-V processor, DOSAGE improved the system's performance and energy by 24.85% and 8.54%, respectively. Compared to a state-of-the-art accelerator generation approach using Microchip's LegUp HLS tool, DOSAGE minimized designer involvement in the accelerator generation process, reduced the number of required accelerators for the applications by 33.33%, and achieved equal or better program coverage.

II. Related Work

The design of application-specific and domain-specific architectures is an active research area due to evolving applications and system architectures. Accelerators have been designed for applications in different domains, like image processing and embedded vision applications [11], cryptography [12], deep learning [13], graph processing [14], etc. These works focus on designing optimal accelerators for specific applications. However, the current design process is ad hoc. That is, the accelerators work for the specific applications for which they are designed; any change in the application or algorithm would require a hardware re-implementation, potentially from scratch. In this paper, our focus is not on optimizing specific accelerators but on developing a methodology to automate the generation and integration of reusable domain-specific accelerators with general-purpose processor systems.

There are a few prior works that are similar to ours. For example, Venkatesh *et al.* [7] presented specialized co-processors that could support multiple general-purpose computations for improving energy efficiency. Their toolchain synthesized *QsCores* by mining for similar code patterns within and across applications. Their analysis showed that multiple data structures could be represented only by four distinct computation patterns, and they integrated the QsCores supporting those patterns with the system. Canis *et al.* [2] presented an HLS tool, *LegUp*, capable of compiling a standard C program to a hybrid processor/accelerator architecture; however, their hybrid flow was constrained to the granularity of functional blocks, which limits the reusability of the generated accelerators. Kumar *et al.* [15] presented an automated toolchain, *Needle*, that could select accelerator-friendly regions within the application codes. They used a new abstraction called 'braids' to merge paths with many common basic blocks to increase the accelerator's code coverage without impacting the hardware complexity.

Our work has two major distinguishing features from prior work. First, we consider a set of input applications and prioritize the recurrent similar blocks so that most of the applications benefit collectively and require a minimal number of accelerators, rather than prioritizing the hotspots in individual applications and generating accelerators for all the applications as in [15]. Second, our work emphasizes the reusability of accelerators for different applications or algorithms, without the need to re-implement the accelerators. To this end, we use a different code granularity, which we call *super-block*, to generate the accelerators, rather than the constrained function-based approach used in prior works [2], [7]. Furthermore, the accelerators in [7] are generated based on similarity in code segments found in the applications' hotspots. In contrast, we generate *super-blocks* for entire applications and use a ranking methodology to select the best *super-block* to be accelerated. Since DOSAGE analyzes entire applications at compile-time, instead of just hotspots, our approach can capture more program similarities across multiple applications.

III. A New Granularity: Super-blocks

In general, domain-specific accelerators can be generated at three application granularities: *basic blocks*, *functions*, and *modules*. *Basic blocks* are the smallest container of sequential instructions. The *function* granularity is at the same level as the functions defined in the high-level program code. *Module* is the largest granularity and can represent a large part of the application. The open-source LLVM compiler, which we use in this work, supports intermediate representations (IRs) in these three granularities.

Most prior accelerators use the *function-based approach* [2] for generating accelerators, since this approach is easier and

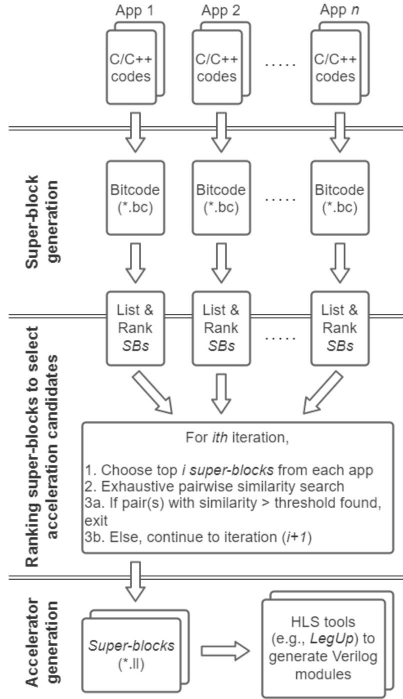


Fig. 3. A high-level overview of the DOSAGE workflow, comprising of *super-block generation, super-block ranking*, and *accelerator generation*

can provide long-term stability. The *functions* can be converted as standard APIs to access the accelerators. However, programmer efforts are needed to modify existing program codes to access the accelerators using new APIs. DOSAGE aims to be programmer-agnostic; as such, the function-based approach is not a good fit.

Using the basic block granularity to generate accelerators would seem like a good fit for our work. However, we found the overhead of this granularity to be prohibitive due to the large number and small size of the generated accelerators using this approach. By analyzing different applications' basic blocks, we found a balanced approach that involved using a new granularity, which we call *super-blocks* (SB). A super-block is a container of adjacent basic blocks in the control flow graph (CFG), and has exactly one entry and one exit point. The single-entry and single-exit condition allows super-blocks to be self-contained units having a deterministic instruction flow, but without the strict constraint of sequential instructions as in a basic block. Thus, super-blocks enable more flexibility with similar analyzability to basic blocks. Super-blocks will generally vary in size; they will typically be larger than basic blocks, but may be smaller or larger than a function.

Fig. 2a illustrates a CFG for a sample arbitrary application. The containers AA, BB, BB1, BB2, and CC are the basic blocks; multiple super-blocks can be generated from the same CFG (in practice, workloads can have hundreds of super-blocks). For example, the highlighted portions of Fig. 2b illustrate two valid super-blocks that can be generated from the CFG. On the other hand, the super-block highlighted in Fig. 2c is invalid since there are two exit points defined (BB to BB1 and AA to CC).

IV. DOSAGE WORKFLOW

Fig. 3 depicts a high-level overview of the DOSAGE workflow. The input is a set of applications written in a high-level programming language (e.g., C/C++). The user has an option to specify each application's *persistence*, which we define as how frequently the application is run compared to other applications within the domain. DOSAGE defaults to an equal persistence for all the applications. The DOSAGE workflow consists of three stages: *super-block generation*, *super-block ranking*, and *accelerator generation*.

Super-block generation: In the first step, using LLVM analysis passes, DOSAGE generates a list of super-blocks for each application by first generating the basic blocks and CFGs using the LLVM compiler tool to obtain the intermediate representations (IR). The applications are compiled using the `-emit-llvm` LLVM flags, which generates an LLVM IR bitcode file (i.e., in the `.bc` format). DOSAGE extracts an LLVM IR instruction-level CFG and a basic block list—each node in the CFG is a basic block—from the bitcode file. To generate a list of all super-blocks, the basic blocks are recursively merged with parent/child nodes and checked for their validity to ensure that each super-block only has one entry and one exit point. Once the list of all super-blocks is generated, DOSAGE then ranks and prioritizes them in the super-block ranking stage.

Super-block ranking: Since several super-blocks—potentially hundreds—can be generated in the super-block generation step, the ranking process is required to prune the generated super-blocks. The super-block ranking process involves a simple ranking heuristic to prioritize the generated super-blocks based on each super-block's provided benefit for hardware acceleration. The ranking heuristic is a two-step process. First, the super-blocks within each application are ranked, and then the ranked super-blocks are compared for similarity across all applications.

The super-blocks are ranked based on three parameters: *execution frequency* (i.e., loops), *the types of instructions* (e.g., divide, mod, and multiply instructions), and *super-block size* measured by the number of instructions. Super-blocks containing code regions with higher execution frequency are ranked higher to ensure that the accelerator is frequently utilized. Complex multi-cycle instructions, like divide, mod, and multiply, are also prioritized for acceleration, since they are typically inefficient in traditional pipelines. The super-block size is considered to minimize the overhead of interfacing the accelerator. Thus, for example, in Fig. 2b, the super-block on the right will have a higher priority than the left super-block. The three parameters have equal weightage in determining the super-block ranking within each application.

Since the focus of DOSAGE is to generate domain-specific accelerators, the highly ranked super-blocks that are common across different applications would be prioritized for hardware acceleration. As depicted in Fig. 3, the process of finding the common blocks follows an iterative heuristic. The highest-ranked super-blocks for the applications are compared for

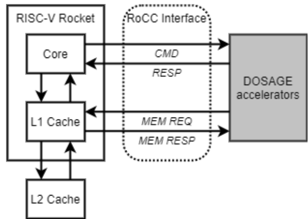


Fig. 4. System overview of the processor in our case study

similarity based on a designer-specified similarity threshold. To measure similarity, DOSAGE compares the CFGs of the super-blocks to find how similar they are in terms of their compositions. This threshold can be varied based on the designer's needs or system resource constraints. A higher threshold would imply stricter similarity requirements, which may result in smaller super-block sizes and fewer common super-blocks, while a lower threshold would increase the super-block sizes at the cost of introducing more uncommon application-specific blocks into the mix of super-blocks. If the similarity criterion is not matched in the first iteration, subsequent iterations compare lower-ranked super-blocks (i.e., for n^{th} iteration, top n-ranked) until a super-block satisfying the similarity threshold is found.

Accelerator generation: When the super-block acceleration candidates are determined, the accelerator(s) are then generated through the traditional high-level synthesis process (e.g., using LegUp, Vivado HLS, etc.). A few modifications may be required to ensure input validity to the selected HLS tool. For instance, in this work, we used LegUp, which can only generate accelerators from inputs that are at the function granularity. However, since DOSAGE is a super-block-based methodology, we modified the LLVM IR (using the `llvm-extract` tool) to wrap the super-blocks in order to emulate a function and provide valid inputs to LegUp. Note that this modification may not be required for other HLS tools.

V. Case Study and Experimental Setup

To rigorously evaluate and exemplify the benefits of DOSAGE, we opted to use a set of diverse real-world applications written by different programmers. Therefore, we chose the applications from the HERMIT benchmark suite [10] to illustrate the effectiveness of the DOSAGE workflow. The benchmark suite comprises of Internet of Medical Things/healthcare applications ranging from wearable device applications to compute-intensive signal processing applications. For the purpose of this work, we classify the HERMIT applications into two domains: *ECG-based applications* and *Image & Signal processing applications*. While the applications were not classified as such in [10], we found the classification to be instructive for our analysis. Table I summarizes the HERMIT benchmarks, and our classification of the applications. For brevity, we omit the details of the benchmarks and direct readers to [10] and the associated Github repository for the description and code of the benchmarks. We did not modify any of the code for our experiments.

We built DOSAGE on top of Clang-LLVM (v3.8), which we used to compile the applications, generate the super-blocks,

TABLE I
Summary of accelerators & kernels

SB-based accelerators	Corresponding FBAs	HERMIT applications
ECG-based applications		
SB1	*fn(ecgsignal)*	*activity*, *apdet*, *hrv*, *sqrs*, *wabp*
SB2	*fn(sqr)*, *fn(sqrt)*	*activity*, *apdet*, *hrv*
SB3	*fn(sin)*	*apdet*, *hrv*
Image & Signal processing applications		
SB4	*fn(mac_a1)*	*imghist*
	fn(mac_a2)	*iradon*

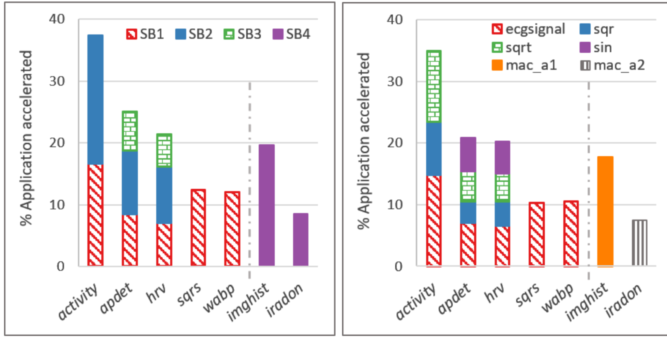
and perform the ranking heuristic described in Section IV. The output of this process is an LLVM intermediate representation (`*.ll` files) of the code blocks for which hardware acceleration would be the most beneficial. We used the LegUp HLS tool (v8.1) to generate the Verilog modules from the LLVM IR. Note that DOSAGE does not specifically depend on LegUp; any other HLS tools capable of using LLVM IR inputs can be used. We chose LegUp since it natively supports the LLVM IR inputs.

Fig. 4 depicts a high-level system overview of our case study. To represent the base processor, we used a RISC-V-based Rocket-core processor. The processor comprises of a single-core 64-bit processor, featuring 16 KiB, 4-way set-associative L1 instruction and data caches, and an off-chip single bank 512 KiB, 8-way set associative L2 cache. For comparison to the state-of-the-art, we also generated accelerators for the HERMIT applications using the function-based approach [2] and compared them to the DOSAGE-generated domain-specific accelerators. The accelerators were connected as custom co-processors to the RISC-V processor using the Rocket Custom Co-processor (RoCC) interface. We implemented the base processor, DOSAGE-generated accelerators, and function-based accelerators (FBAs) using a Nexys4-DDR FPGA to evaluate the performance, energy, and area implications.

VI. Experimental Results

This section discusses the experimental results of generating hardware accelerators using DOSAGE for the HERMIT applications. We also analyze the DOSAGE-generated accelerators in comparison to FBAs generated using prior work [2] within the context of the baseline processor.

Generated super-blocks: In total, the first DOSAGE step generated the following number of *super-blocks* for the seven applications: *activity:* 564; *apdet:* 606; *hrv:* 591; *sqrs:* 511; *wabp:* 485; *imghist:* 693; and *iradon:* 739. After the super-block ranking step, using a sample similarity threshold of 80%, DOSAGE pruned the super-blocks to just four candidate super-blocks for all the HERMIT benchmarks (three for the ECG-based applications, and one for the signal processing applications). For simplicity, we refer to these super-blocks as SB1, SB2, SB3, and SB4 (see Table I). To understand the super-blocks' composition, we analyzed them and identified their corresponding functions in the applications' high-level codes, as depicted in Table I. For the ECG-based applications



(a) DOSAGE-based accelerators (b) FBAs

Fig. 5. Program coverage of hardware acceleration using (a) DOSAGE-based accelerators (b) FBAs (prior work). DOSAGE achieves equal or better program coverage using fewer accelerators.

(*activity*, *apdet*, *hrv*, *sqrs*, and *wabp*), the best candidate for hardware acceleration (and the highest-ranked common super-block by DOSAGE) was SB1, which was common to five HERMIT benchmarks. The second best candidate for hardware acceleration was SB2, which was common to three HERMIT applications and consisted of the math functions *sqr* and *sqrt*, while SB3 was common to two HERMIT applications. For the image and signal processing applications, *imghist* and *iradon*, the sole generated super-block SB4 comprised of multiply and accumulate (MAC) operations. To select SB1, SB2, and SB3, DOSAGE required 31 iterations in the super-block ranking step, while it required 14 iterations to select SB4.

Static analysis of program coverage: In total for all seven benchmarks, four DOSAGE-based accelerators were required as opposed to six FBAs (Table I), representing 33.33% fewer accelerators. More FBAs were required due to the programming of the different applications, wherein functions with similar computations were coded in different ways. For instance, in *imghist* and *iradon*, functions with similar MAC loop operations required two different FBAs, whereas DOSAGE consolidated the operations into one accelerator, SB4. Even though this diversity is an artifact of modern-day programs, DOSAGE mitigates this limitation by searching for common code blocks and computations at a lower level without being constrained to the function-level granularity.

Ideally, the DOSAGE-generated accelerators must provide similar program coverage to the state-of-the-art accelerators. Thus, we analyzed the amount of each application covered by the DOSAGE-based accelerators compared to FBAs. Fig. 5 depicts the percentage of program coverage using the DOSAGE-generated accelerators (Fig. 5a) and the corresponding FBAs (Fig. 5b). We measured the program coverage as the percentage of the LLVM IR lines accelerated by the different approaches. On average, DOSAGE accelerated an extra 2.30% and 1.41% of the program for the ECG-based and signal processing applications, respectively, compared to FBAs. DOSAGE achieved equal or better coverage than the function-based approach, while also reducing the total number of accelerators. For instance, 37.42% of *activity*'s code could be accelerated with two accelerators using DOSAGE, compared to three FBAs. The amount of code that could be accelerated was constrained by the application characteristics. For instance, *iradon* has the least possible application acceleration at 8.42% due to the application's characteristics (e.g., large code size, limited ILP, etc.) [10].

TABLE II
IMPLEMENTATION OVERHEAD OF DOSAGE ACCELERATORS COMPARED TO FBAS (PRIOR WORK).

FBA	Area (LUTs)	Power (W)	DOSAGE	Area (LUTs)	Area Gain	Power (W)	Power Gain
fn(ecgsignal)	1641	0.057	SB1	1692	-3.11%	0.056	1.75%
fn(sqrt)	3443	0.131	SB2	6533	-3.48%	0.230	0.86%
fn(sqr)	2870	0.101					
fn(sin)	1816	0.065	SB3	1876	-3.30%	0.066	-1.54%
fn(mac_a1)	1175	0.046	SB4	1239	47.79%	0.047	49.46%
fn(mac_a2)	1198	0.047					

Overhead analysis: Table II summarizes the post-implementation results of the different hardware accelerators. Overall, DOSAGE did not increase the overhead compared to the FBAs, even though DOSAGE accelerators covered more operations. For instance, even though SB2 consumed the largest area among all super-blocks, it was only marginally larger (3.48%) than the combination of the two equivalent accelerators in the function-based approach. On the other hand, SB4 substantially reduced the area and power by 47.79% and 49.46%, respectively, compared to the equivalent FBAs. In addition, the critical paths of the DOSAGE-generated accelerators were similar to those of the FBAs.

Dynamic analysis of speedup and energy savings: Fig. 6 summarizes the execution time speedup, energy savings, and area overhead by incrementally adding DOSAGE accelerators (e.g., first just SB1, then SB1 + SB2, etc.) and the equivalent FBAs to the baseline processor. The priority order for the accelerators for ECG-based applications, determined by DOSAGE, was: SB1, SB2, and SB3. The equivalent priority for the FBAs was: *fn(ecgsignal)*, *fn(sqrt)*, *fn(sqr)*, and *fn(sin)*. For the signal processing applications, the priority order for the two FBAs based on program coverage was: *fn(mac_a1)* and *fn(mac_a2)*.

In a resource unconstrained scenario, where *all* the generated accelerators were included in the system, DOSAGE outperformed the FBAs in both execution time and energy consumption. DOSAGE achieved 30.43% and 4.53% execution time speedups over the baseline processor and the function-based approach, respectively. DOSAGE reduced the energy by 4.23% and 6.20% compared to the base and the function-based approach, respectively, without introducing any area overhead compared to the FBAs.

We observed that as the number of accelerators increased in the system, the acceleration efficiency decreased due to the increase in area overhead. For example, in Fig. 6, using only SB1 improved the system's performance (by 16.71%) and energy savings (by 7.49%) for just 4.48% area overhead compared to the base processor. In contrast, when three accelerators (SB1, SB2, and SB3) were used, the system's performance and energy savings improved by 30.43% and 4.23%, respectively, for a high 24.77% area overhead. Note

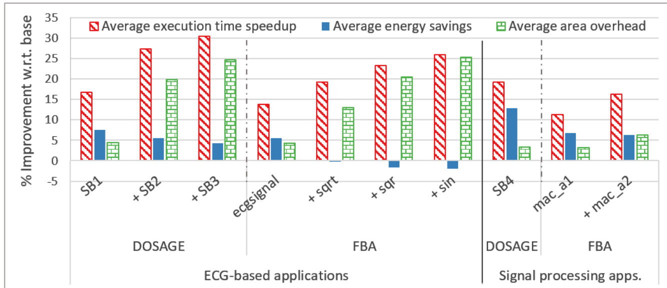
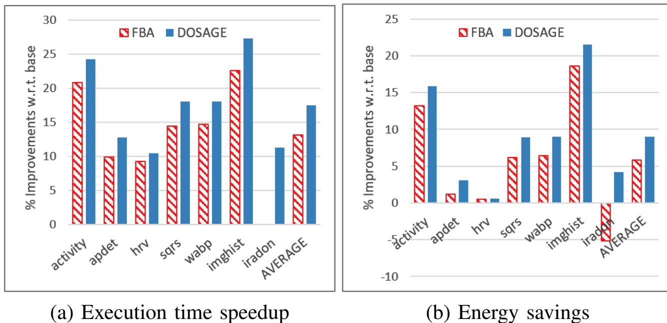


Fig. 6. Average execution time speedup, energy savings, and area overhead results for HERMIT applications using DOSAGE-based accelerators and FBAs compared to the base RISC-V processor. Accelerators are incrementally added to the system. For example, '+SB2' means both SB1 and SB2 are added; '+SB3' means SB1, SB2, and SB3 are added; etc.



(a) Execution time speedup (b) Energy savings

Fig. 7. Execution time speedup and energy savings for HERMIT applications using single DOSAGE-generated accelerators (SP1 and SP4) and FBAs (*fn(ecgsignal)* and *fn(mac_a1)*) compared to the base RISCV processor

that prior work incurred a higher area overhead than ours. Interestingly, the energy savings reduced with an increase in the number of accelerators. This reduction was due to the under-utilization of the accelerators, since the accelerators were not equally useful to all the applications. As such, the accelerators continued to consume power even when not in use. This overhead can be mitigated by switching the accelerators into a low-power state when not in use, but this optimization is outside the scope of this paper. For the FBAs, we observed an energy increase with more accelerators. While an unrestricted increase in accelerators is not an ideal design, these results illustrate the efficiency of DOSAGE compared to the function-based approach.

To further illustrate the efficiency of DOSAGE, we explored a more resource-constrained scenario wherein the system could only be designed with a limited number of accelerators. DOSAGE chose the SB1 and SB4 super-blocks as the ideal acceleration candidates for the ECG-based and image/signal processing domains, respectively. Fig. 7 shows the results for the resource-constrained scenario using DOSAGE-generated accelerators and the equivalent FBAs ((*fn(ecgsignal)* and *fn(mac_a1)*). Compared to the function-based approach, DOSAGE improved the average execution time and energy by 4.34% and 3.18%, respectively, for all the applications. In the resource-constrained scenario, *iradon* could not be accelerated by the *fn(mac_a1)* FBA; however, DOSAGE was able to speedup *iradon* by 11.25% with 9.38% energy savings. Importantly, as previously alluded to, DOSAGE also substantially reduced the area overhead in this scenario.

VII. Conclusion

In this work, we presented DOSAGE, a programmer-agnostic LLVM compiler infrastructure-based methodology for generating domain-specific accelerators for resource-constrained computing systems. The DOSAGE methodology uses a simple ranking heuristic to prioritize the recurrent and similar code blocks from a set of applications, and identifies the code blocks that would benefit the most from hardware acceleration. Experiments using benchmarks in two application domains showed that DOSAGE reduced the required number of domain-specific accelerators and improved the system's performance and energy efficiency compared to prior work. For future work, we will explore DOSAGE in multi-processor systems and include runtime execution characteristics to improve the accelerator ranking heuristic.

References

[1] T. Feist, "Vivado Design Suite," White Paper, Xilinx, Inc., pp. 1–14, Apr. 2012. [Online]. Available: http://xilinx.eetrend.com/files-eetrend-xilinx/download/201205/2440-4537-4wp416-vivado-design-suite.pdf

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, and J. H. Anderson, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[3] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proc. IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[4] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *Proc. IEEE 15th Int. Symp. High Performance Computer Architecture (HPCA'09)*, Mar. 2009, pp. 313–322.

[5] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *Proc. 2016 Int. Conf. Field-Programmable Technology (FPT'16)*, Dec. 2016, pp. 77–84.

[6] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, pp. 101 – 119, 2018.

[7] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO'11)*, Dec. 2011, pp. 163–174.

[8] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, p. 48–57, Jun. 2020.

[9] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA'16)*, 2016, pp. 27–39.

[10] A. Limaye and T. Adegbija, "HERMIT: A Benchmark Suite for the Internet of Medical Things," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 4212–4222, Jun. 2018.

[11] B. Sun, L. Yang, P. Dong, W. Zhang, J. Dong, and C. Young, "Ultra power-efficient cnn domain specific accelerator with 9.3tops/watt for mobile and embedded applications," 2018.

[12] L. Wu, C. Weaver, and T. Austin, "CryptoManiac: a fast flexible architecture for secure communication," in *Proc. Annu. Int. Symp. Computer Architecture*, 2001, pp. 110–119.

[13] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO'16)*, 2016, pp. 1–13.

[15] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging Program Analysis to Analyze and Extract Accelerators from Whole Programs," in *Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA'17)*, Feb. 2017, pp. 565–576.