Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey

NUNO PAULINO, JOÃO CANAS FERREIRA, and JOÃO M. P. CARDOSO, INESC TEC and Faculty of Engineering of the University of Porto

The breakdown of Dennard scaling has resulted in a decade-long stall of the maximum operating clock frequencies of processors. To mitigate this issue, computing shifted to multi-core devices. This introduced the need for programming flows and tools that facilitate the expression of workload parallelism at high abstraction levels. However, not all workloads are easily parallelizable, and the minor improvements to processor cores have not significantly increased single-threaded performance. Simultaneously, Instruction Level Parallelism in applications is considerably underexplored. This article reviews notable approaches that focus on exploiting this potential parallelism via automatic generation of specialized hardware from binary code. Although research on this topic spans over more than 20 years, automatic acceleration of software via translation to hardware has gained new importance with the recent trend toward reconfigurable heterogeneous platforms. We characterize this kind of binary acceleration approach and the accelerator architectures on which it relies. We summarize notable state-of-the-art approaches individually and present a taxonomy and comparison. Performance gains from 2.6× to 5.6× are reported, mostly considering bare-metal embedded applications, along with power consumption reductions between 1.3× and 3.9×. We believe the methodologies and results achievable by automatic hardware generation approaches are promising in the context of emergent reconfigurable devices.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Hardware → Hardware accelerators**; *Application specific instruction set processors*;

Additional Key Words and Phrases: Binary acceleration, runtime instruction traces, hardware synthesis

ACM Reference format:
Nuno Paulino, João canas Ferreira, and João m. p. Cardoso. 2020. Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey. *ACM Comput. Surv.* 53, 1, Article 6 (February 2020), 36 pages.
https://doi.org/10.1145/3369764

1 INTRODUCTION

Until approximately a decade and a half ago, it was straightforward to scale the computing power of processors. By scaling down the technology node, it was possible to decrease the supply voltage, increase the operating frequency, and maintain heat dissipation per unit area constant. This behavior is known as Dennard scaling [26], but as Figure 1 illustrates it broke down at around 2005.

This work was supported by the PEPCC project, "PTDC/EEI-HAC/30848/2017," financed by Fundação para a Ciência e Tecnologia (Portuguese Fundation for Science and Technology).

Authors' addresses: N. Paulino, J. Canas Ferreira, and J. M. P. Cardoso; emails: {nuno.m.paulino, joao.c.ferreira, joao.paiva.cardoso}@inesctec.pt.

0360-0300/2020/02-ART6 $15.00
https://doi.org/10.1145/3369764

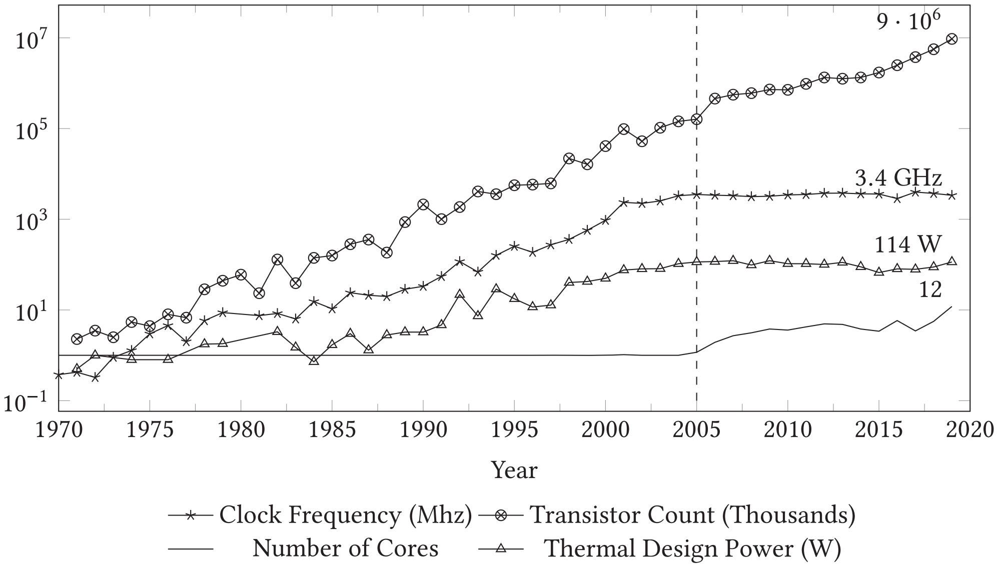


Fig. 1. Trends for desktop and server grade processors throughout the past 50 years, built from 950 data points from CPU DB [23] and Intel's and AMD's product pages.

The four trend lines shown span the past 50 years of microprocessor technology. We first constructed a dataset totaling approximately 3,500 entries. Then we selected, at most, the 30 processors with the highest operating frequency for each year and computed the respective averages of the four parameters. This gives us a look at the upper bounds of processor technology in regard to computing power. We can observe the stagnation of clock frequency, which came about due to current leakage effects at increasingly small scales. Supply voltage could no longer be reduced as before to prevent the increase of heat dissipated per unit area and the resulting thermal runaway. To achieve further performance increase, thread level parallelism was explored by development of multi-core processors. The replication of cores, together with larger caches, is responsible for the continued increase in transistor count. However, to maintain heat dissipation per unit area constant, most transistors on the device are powered down at any given time. The term *dark silicon* refers to this underutilization of transistors [88]. Together with the fact that performance improvements from parallelization are subject to Amdahl's law, the scalability of multi-core seems compromisd on a long term [28, 29].

However, studies show that *Instruction Level Parallelism* (ILP) in single-core workloads has never been fully explored [33, 95], despite existing compilation techniques and architecture features such as multiple-issue pipelines, vectorization, software pipelining [52], and *Very Long Instruction Word* (VLIW) or Superscalar processors. So, in the lack of a technological improvement to restore a sustainable scaling of processing performance relative to power consumption, more specialized architectures may provide a complementary solution by exploiting unexplored ILP.

An effective way to achieve this is to design custom hardware on a per-application basis to compliment or enhance present day general purpose processors. However, this requires specific hardware expertise and determining which parts of the application are suited for implementation on custom hardware, e.g., a *Hardware/Software* (HW/SW) partitioning effort [89]. Hardware design is a lengthy and difficult task, and continued revisions to the application are hindered by the possibility of having to re-design the hardware. This is exacerbated as the partitioning step is typically iterative in attempts to improve the HW/SW solution. *High Level Synthesis* (HLS) [35] approaches address the issue of the hardware design by automatically generating hardware descriptions from source code such as *C/C++* or OpenCL [12, 55]. In traditional processors, performance is improved by exploiting ILP via multiple-issue pipelines and by applying techniques like software pipelining [52]. HLS approaches attempt to improve upon this by generation of pipelined datapaths whose stages execute independent operations concurrently.

This article focuses on work with a similar design philosophy but that attempts to rely only on binary level information, and reduced developer intervention, to generate customized hardware co-processors/accelerators. Candidate portions of the target application's binary are detected and translated into an accelerator configuration or description for synthesis. We explain binary translation and explore the features and architecture of accelerator designs in the following sections, thoroughly discussing the latter in particular. Finally, we organize said features of both these aspects in two corresponding taxonomies. The main contributions of this article are as follows:

- An explanation of binary translation targeting hardware accelerators (Section 2)
- A thorough explanation of accelerator architectures (Section 3)
- Detailed descriptions of individual approaches we find to be most notable (Section 4)
- A final comprehensive taxonomy and comparison of all approaches (Section 5)

Several literature reviews focus on aspects of reconfigurable architectures employed as accelerators, such as specific topologies for processing elements [5] or interconnection schemes [98], or classifications and comparisons in general [15, 20, 45, 82]. In Reference [36], HW/SW partitioning and custom/reconfigurable architectures are presented in the context of Instruction Set Extension approaches. This survey categorizes and presents the steps involved in retargeting binaries to hardware accelerators, along with a characterization of these accelerator architectures.

This article is organized as follows. Section 2 elaborates on the several aspects of the binary analysis and translation steps in the context of binary acceleration as we have defined it. Section 3 outlines the general characteristics of all reviewed accelerators. In Section 4, we present further details on each reviewed approach. Section 5 provides a summary and classification charts according to the taxonomies we propose, and Section 6 concludes the article.

2 BINARY TRANSLATION TO HARDWARE ACCELERATORS

Binary translation is a form of recompilation wherein a given application, compiled for an *Instruction Set Architecture* (ISA), is partially or wholely transformed into binary code for another ISA or into the same ISA with added modifications or optimizations [1]. Translation can be performed over the compiled binary code, i.e., directly over compiler output, or over the executed instruction stream. Example applications of the former include providing compatibility between ISA without resorting to re-compilation [6, 16] or code instrumentation [53, 104]. Notable examples of the latter are the *Just-in-Time* (JIT) compilation employed in the Java Runtime Environment, tools such as Valgrind [64] and virtualization [81]. During the translation process, several optimization techniques can be applied to improve performance, e.g., by taking advantage of specific micro-architectural capabilities of the target device [42] or by relying on profiling information to re-organize code and reduce branch misprediction [24]. Work also exists in translating RISC code to VLIW [27], on implementing the binary translation process in hardware [78], and on combining static and dynamic binary translation for greater efficiency [96]. A recent survey extensively covers the past five decades of binary translation techniques and approaches [97].

The approaches reviewed in this article are based on a specific type of binary translation, which targets selected portions of an application binary, leaving the remainder to be executed in the host processor. The selected binary is translated into configurations for specialized accelerator

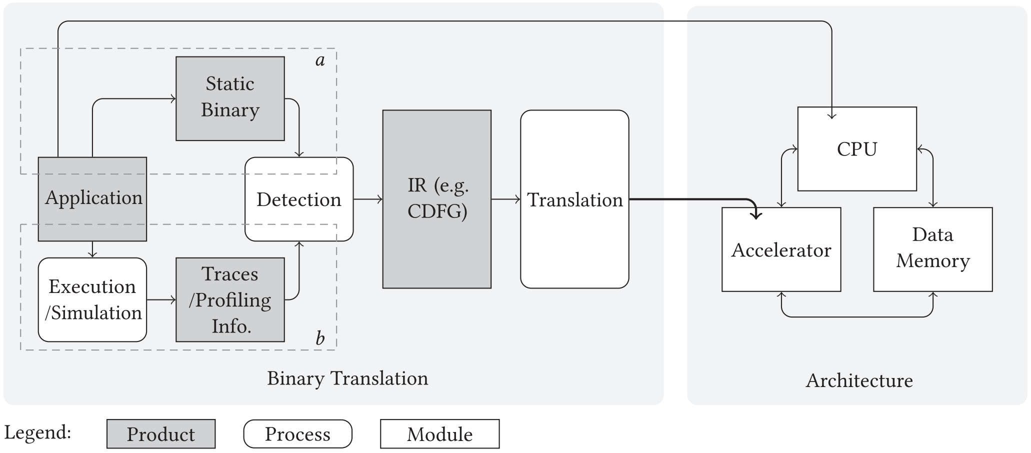


Fig. 2. High-level generic representation of binary translation flow into custom hardware ((*a*) flows based on complied binary and *(b*) flows based on binary traces).

hardware or into hardware descriptions for later synthesis. Figure 2 represents a high-level view of this process. The first step is to take either the application binary, or execution traces taken from a simulator or actual host processor, and automate the detection of the *binary segments* to accelerate. These segments are whatever portions of binary code in the application whose execution is migrated to the accelerator. Each approach relies on a different type of segment, but all fit in the following general definition:

> **Binary segment:** list of binary instructions whose execution is sequential and may contain one or more control instructions that end or discard the execution of the list

These segments are migrated by a translation process to accelerator hardware, whose purpose is to exploit latent ILP by specialization of its *Functional Unit*s (FUs) and their interconnections. That is, this step generates and/or configures the accelerator module by producing a full hardware description, customizing a template, or performing instruction set extension on the host processor. Generating control words for a local configuration memory of the accelerator and integration with the host processor are also common steps. The resulting architecture is typically composed by a host processor, a shared data memory, and the accelerator. The type of binary segment and the accelerator architecture are related as the former is represented in such a way as to be executable in the proposed accelerator, or the latter is designed in such a way as to execute the proposed binary segment. Translation is usually performed without relying on source code, manual HW/SW partitioning, and/or hardware design efforts. We categorize approaches presenting these features as *binary acceleration* approaches, which we define as:

> **Binary acceleration:** post-compile acceleration of applications via automatically generated accelerator hardware, or accelerator configurations, from application binary

There are drawbacks on relying on binary code relative to high-level code, as HLS approaches do. For instance, high-level code constructs are absent (e.g., *for*, *if-else*); it is difficult to perform memory disambiguation, to unroll or fission loops, or to determine memory access patterns or trip counts of loops. Also, operations on certain data types (e.g., *double*) may be replaced with

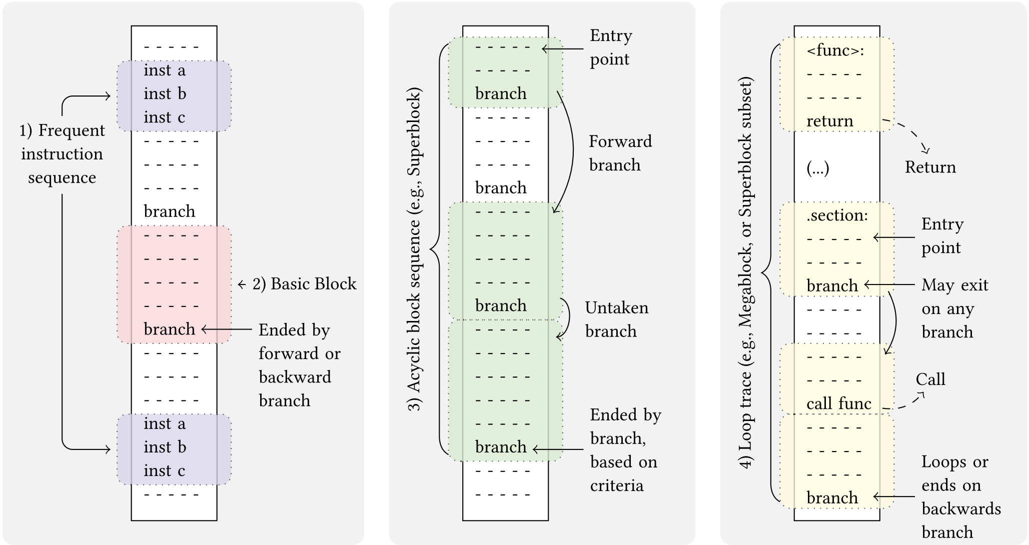

- - - - -
inst a
inst b
inst c
1) Frequent instruction sequence
branch
2) Basic Block
Ended by forward or backward branch
3) Acyclic block sequence (e.g., Superblock)
Entry point
Forward branch
Untaken branch
Ended by branch, based on criteria
4) Loop trace (e.g., Megablock, or Superblock subset)
<func>:
return
Return
(...)
.section:
Entry point
May exit on any branch
call func
Call
Loops or ends on backwards branch

Fig. 3. Four types of binary segments than can be extracted from sequences of binary instructions.

software emulation subroutines on targets without a *Floating Point Unit* (FPU), and the same applies to operations such as the modulus, further hindering effective analysis. In contrast, manually partitioning an application requires an understanding of its tasks, how they exchange data, and their influence on the execution time and power/energy consumption, in addition to understanding how the source-code relates to the generated hardware and its efficiency [3, 25, 55]. Also, binary code is more easily parsed and interpreted compared to source code. For the same reason, it is also tractable enough to allow for an embedded approach to binary analysis and translation to be viable. Finally, relying on binary for acceleration promotes portability, allows for the same binary to be accelerated in different ways, depending on the underlying acceleration system and offloads design effort from the software designer to the system.

Given this terminology, we identify the following distintive characteristics of the binary translation process, which comprise the classification taxonomy employed in Section 5 to classify each approach's method of binary translation:

- The type of accelerated binary segment;
- The method for binary segment detection;
- The method for binary segment translation;
- Need for modifications to the application binary;
- The type of targeted accelerator architecture.

The following paragraphs further explain these points by exemplifying how several types of binary segments can be constructed and are used by binary acceleration approaches, the methods for their detection, and translation. Additionally, a summary for these features and respective values is shown at the end of this section, in Table 1.

2.1 Types of Binary Segments

How different types of segments can be constructed from an application's binary is illustrated in Figure 3. Three synthetic excerpts of binary code are shown. Each dashed line represents any

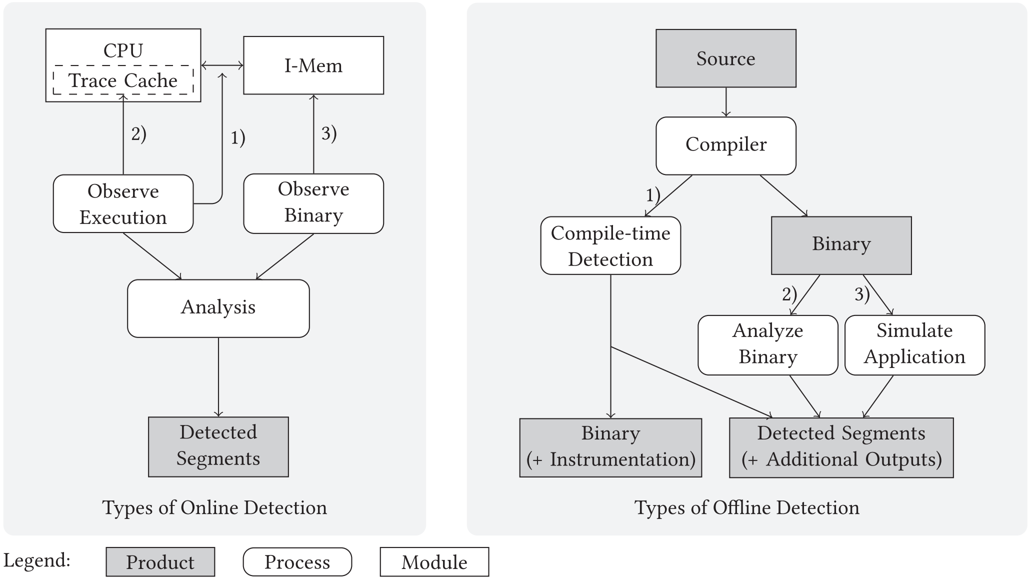


Fig. 4. Different methods for detection of binary segments.

instruction of the target instruction set architecture other than branches. Only instructions relevant to the construction of the segment are explicitly named.

Frequent instruction sequences and *Basic Block*s (BBs) are illustrated on the left-hand side of Figure 3. The former are simply short sequences of instructions that appear frequently, and the latter are sequences of instructions of any size containing, at most, a single *branch* operation that ends the block. Both of these segment types can also be extracted from binary traces, based on execution frequency rather than ocurrence frequency. Another type of segment represents acyclical execution flow, i.e., an execution path through the code such that no instruction is executed more than once. The center of Figure 3 shows an example Superblock [46], which is constructed from a sequence of forward-jumping BBs and profiling information on branch frequency. Finally, some segments can represent loop bodies or iterations of a loop path and contain control instructions to terminate execution. The right-hand side of Figure 3 shows such a segment, called MegaBlock [10]. The execution of the code is followed starting from a single entry point through taken branches either backward of forward and ends at a backward branch that loops back to the entry point.

There are minor per-approach variations to these segments, for instance, maximum size (e.g., number of instructions) of the segment [19, 77], presence of instructions unsupported by the accelerator [17, 79], total number of inputs/outputs into the sequence [13, 68], the type of stride [58] in the case of loops, or support for conditional execution, i.e., multiple paths [67].

2.2 Segment Detection

These segments can be detected online (at runtime) or offline (at design-time), although simpler segments are more amenable to an online detection and later translation. Figure 4 shows three possible methods for online detection of segments and likewise for offline detection. The detection process may be aided by prior binary instrumentation to mark the code ranges to consider [17].

Online detection could resort to having auxiliary hardware read the main processor's code memory (*3*) to analyze the binary. Alternatively, execution traces, and therefore hot regions of code and other information, such as branch frequency, can be retrieved by observing a trace cache or the

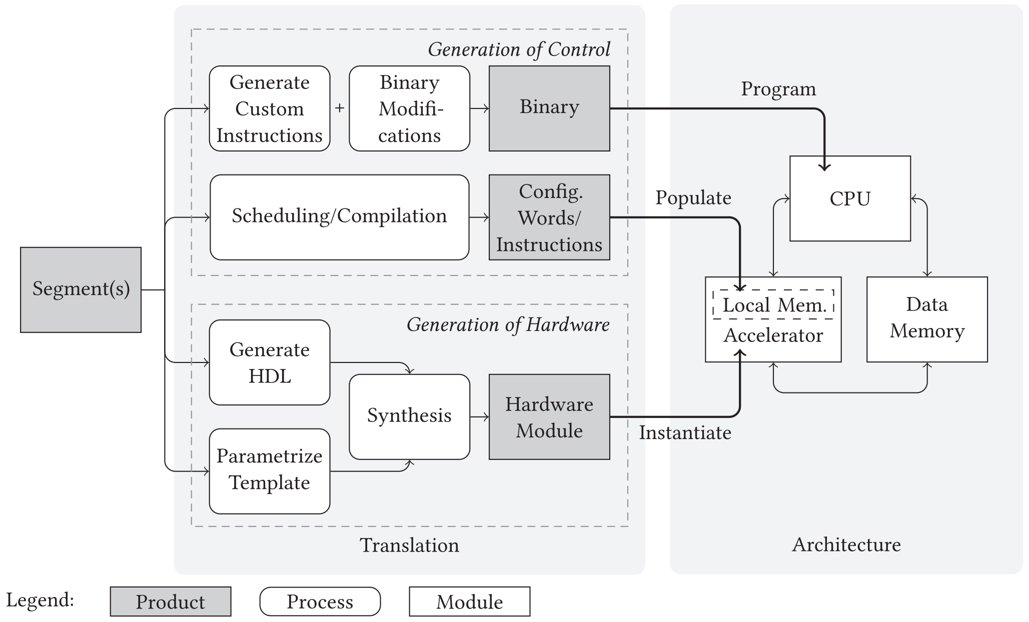


Fig. 5. Processes that may be involved in a translation step.

decode stage (*2*) [19] or by monitoring the instruction bus or program counter (*1*) [58]. Performing detection online may limit it to short segments due to time or processing power constraints but provides greater transparency. Offline detection benefits from few constraints regarding memory, processing power, or overhead and can extract lengthier binary segments. To implement offline detection, some approaches integrate the detection into the compiler [17, 32, 40, 44] (*1*), while others analyze the application post-compilation either statically [70] (*2*) or by simulation [67, 73] (*3*). Additional outputs, such as profiling information may also be produced [73].

2.3 Segment Translation

The translation step involves retargeting the detected segments to the proposed accelerator architecture. Generally, ILP is first exposed in the segment by representing it as an intermediate *Control and Dataflow Graph* (CDFG). After this, each approach implements translation differently, depending on the type of binary segment and accelerator architecture. We identify two generic parts of this process: the generation of control (i.e., configuration words, instructions, or schedules) and the generation or specialization of the accelerator. Figure 5 shows a generalized representation of these tasks, some of which can be performed in conjunction.

Generating the control for the accelerator is essentially the assignment of the operations in the segments to each available FU of the accelerator, which is why this process differs per approach based on the accelerator. For instance, the generation of custom instructions that extend the host ISA is usually applied to small binary segments such as non-cyclical short sequences or short BBs. This method often occurs in approaches that rely on small accelerators integrated into the processor. The binary is modified by replacing the segments with one or more custom instructions of lower latency that activate the accelerator [17, 40, 68] and control any of its internal logic [39].

In contrast, for accelerators that execute their segments throughout multiple clock cycles, FUs are used repeatedly to execute several operations of the same segment. The configuration words must control the dataflow of operands and results between the FUs according to the respective

Table 1. Summary of Binary Translation Taxonomy Features and Values

Feature	Key	Value #1	Value #2	Value #3
Segment Type	*A*	Static segments	Acyclic traces	Cyclic traces
Binary Detection	*B*	Static Analysis	Offline Profiling	Online Profiling
Binary Translation	*C*	Compile-time	Post-Compile	Runtime
Binary Modification	*D*	Compile-time	Post-Compile	Runtime/None
Accelerator Type	*E*	Static Design	Template Based	Full Custom Synthesis

CDFGs that represent the segments. The results are wider configuration words, or schedules, which control the activation of each FU and the transit of data between them and/or any existing local memories. This is typically employed to accelerate larger code segments, e.g., segments representing loops, which benefit from reutilization of accelerator resources. A single configuration word might be sufficient to control the context of the accelerator throughout execution [92] or several configuration words may be required to implement a temporal partition of a segments CDFG, i.e., a schedule [73]. Regardless, local memories can be employed to store the configurations/instructions.

Regarding the actual accelerator hardware, some approaches rely on a static accelerator architecture, whose design is determined by an empiric analysis based on a representative set of segments (e.g., based on application domain). This allows for the system to perform detection and translation at runtime due to the simpler tasks involved [92]. Alternatively, the accelerator may be specialized directly as a function of the detected segments by either generating tailored hardware descriptions or by creating an instance based on a template that allows specifying the number and type of FUs and connections between them. Either case requires posterior synthesis via tools, which constrains this degree of specialization to offline implementations. However, like detection, offline translation can benefit from generating more specialized hardware, in combination with loop pipelining [73] and more agressive exploitation of data-parallelism [56].

In summary, we identify three major methods for accelerator generation or configuration: (1) targeting a static pre-designed accelerator architecture [40, 70], (2) specializing an architecture template followed by generation of configuration words, instructions, or schedules [17, 32, 73], and (3) generation of a full custom hardware description [56, 58].

2.4 Summary of Binary Translation Taxonomy

Table 1 summarizes taxonomy on the binary translation aspect of each approach, presenting the possible values we have outlined for each feature (e.g. *A1* signifies the approach relies on binary segments retrieved from static analysis of the binary). We sort the possible values for all features from least to most sophisticated. We consider that the adoption of binary translation for automatic acceleration of code, especially its use as a part of future self-adaptive systems, requires that it does not interfere with compilation flows and does not compromise binary portability. Therefore, we consider that approaches are more sophisticated if they implement as much as possible of the binary translation process at runtime, i.e., the detection, translation, and any binary modification. As for the segment and accelerator type, acceleration potential is greater for larger binary segments, and greater customization of the accelerator more efficiently realizes this potential. The discussion in Section 4 on the notable approaches will classify their respective binary translation method according to these values (e.g. *A1, B2, C2, D2, E3*).

3 CHARACTERISTICS OF ACCELERATOR ARCHITECTURES

An accelerator, as opposed to a general purpose processor, is a hardware device designed to execute a specific workload or family of workloads. Through specialization, greater performance and energy efficiency are achieved, at the expense of versatility. A notable example of hardware accelerators are *Graphics Processing Unit*s (GPUs) and audio processors, widely used in desktop computers. Other application specific accelerators can be found in data centers and services, targeting domains such as databases [43], machine learning [87], or cryptography [11]. Due to more versatile programming abstractions like CUDA [66] and OpenCL [86], GPUs also represent some of the market share throughout these applications [4, 61]. Additionally, *Field Programmable Gate Array*s (FPGAs) have also found use in such domains in recent years due to increasingly more sophisticated and friendly tools and flows based on HLS, including programming through OpenCL [49, 101].

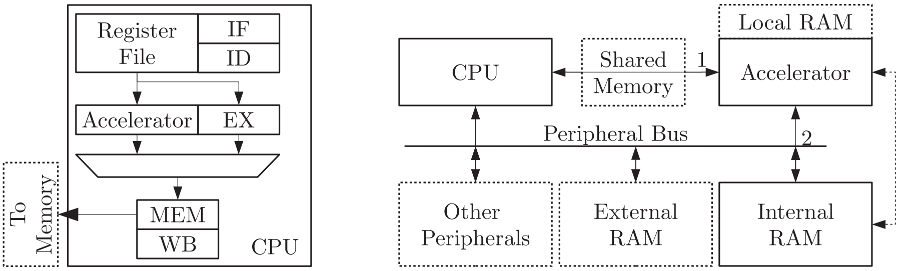
These domains and respective hardware acceleration approaches are large scale and big data systems. The same advantages of hardware specialization on a per-case basis, i.e., for specific workloads of general purpose embedded applications, remain unexplored for smaller scale systems for which the use of GPUs or *Application Specific Integrated Circuit*s (ASICs) is unviable. This is the issue targeted by the approaches summarized in this article, and in this section we present a synopsis of the architecture characteristics of the accelerators they propose. The accelerators are generally composed of a set of coarse-grained FUs, interconnection scheme, and interface to host processor. Local data and instruction memories may also be present. By adjusting the FU layout and interconnection complexity, accelerator design favors either specialization or versatility. The specific details vary in function of the type of targeted binary segment, which accelerators execute to improve execution time and, ideally, power/energy consumption. We outline the following main characteristics of accelerator architectures:

- Type of interface between accelerator and host system;
- *Functional Unit* Arrangement;
- *Functional Unit* Interconnections;
- Operations supported by the *Functional Unit*s;
- Memory access capabilities;
- Execution model of the accelerator.

What follows are explanations of these aspects, demonstrating the general implementation strategies we observed for each axis of this taxonomy, along with respective references. Further details on the most notable approaches included in this section are included in Section 4, and more emphasis on comparisons is included in Section 5. Finally, a summary for these features and respective values is shown at the end of this section, in Table 2.

3.1 Accelerator-Host Interface

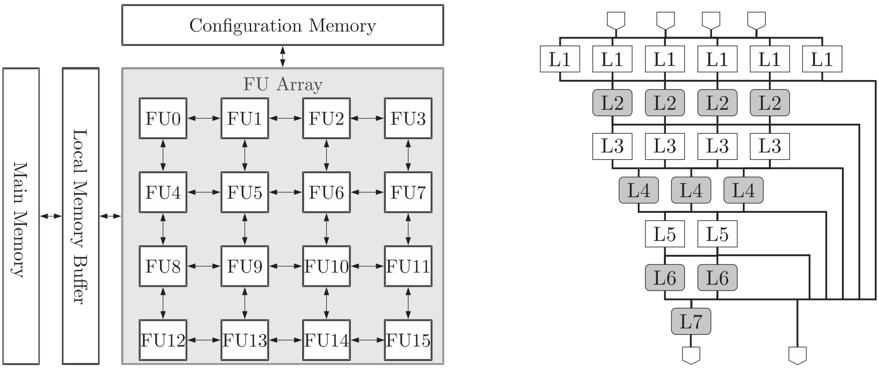
Using an accelerator implies an exchange of information between it, the host processor, and, in some cases, a shared memory. The two major types of interfaces employed are shown in Figure 6. The first couples the accelerator to the processor's pipeline, directly accessing values in the register file [8, 19, 40, 68], as shown in Figure 6(a). This reduces the overhead between the accelerator and the host processor but may make it more difficult to use the accelerator with different hosts. Placing additional logic in the processor's pipeline could also introduce critical path delays. Additionally, multiple concurrent memory accesses by the accelerator become more difficult to implement, since there is no clear way to interface with data memories or to allow concurrent accesses. Alternatively, accelerators may be loosely coupled as peripherals [34, 57, 72, 83], using interfaces such as buses, dedicated links, or shared memory schemes [44, 56, 58, 71, 94], as shown in Figure 6(b). Although these interfaces introduce larger overheads, they avoid intrusive modifications to the host processor. In addition, since the accelerator is an external peripheral, design is less constrained.



(a) Tight coupling into host processor pipeline

(b) Loose coupling via: 1) point-to- point connection; or 2) bus connection

Fig. 6. Different interfaces between host processor and accelerator (optional components in dotted lines).



(a) Mesh arrangement (adapted from [51])

(b) Row-based arrangement (adapted from [19])

Fig. 7. Two *Functional Unit* arrangements and interconnections for accelerators.

3.2 FU Arrangement

The arrangement of FUs in accelerators is either row or mesh based, as exemplified in Figure 7. A mesh arrangement is shown on the left-hand side (Figure 7(a)). This type of design is characterized by FUs that are usually homogeneous and richly interconnected. In most cases, FUs can receive and send operands and results from immediate neighbours, and some FUs may contain their own register files [34, 71]. This type of design scales easily due to its homogeniety and is usually employed in loosely coupled accelerators. A thorough summary of this type of architecture is presented in Reference [45]. Figure 7(b) shows a row arrangement of FUs, where units in the same row execute concurrently and propagate data downward. This type of architecture is common for small accelerators, which can be used as units integrated into the host pipeline, and its constituent FUs are more heterogeneous. As a particular case, some approaches rely on a single row of units [22, 37].

3.3 FU Interconnections

Interconnection capability especially influences the performance of an accelerator. A richer interconnection scheme typically allows for a better use of the available computing resources. In mesh arrays, supporting connections to neighbour FUs is the least complex scheme, followed by diagonal connections and longer lines connecting non-neighbours. Row arrays are designed with the forward (i.e., downward) propagation of data in mind and thus rely on components ranging from specialized multiplexers to full crossbars. Connections may also span multiple rows [17, 68, 72] or may implement backward connectivity. However, more complex interconnects require more control information, introducing configuration overhead and storage overhead for the configuration words/instructions. Elements like crossbars and multiple buses may also introduce delays in critical paths and require considerable resources. Binary segments are often first represented as CDFGs during translation. Since nodes have different connectivity, resources in a very high connectivity mesh may be under-utilized. Instead, connectivity can be tailored to the set of segments or an application domain. Different interconnection topologies for meshes are evaluated in References [63, 93, 98]. Likewise, quantitative analyses can determine connectivity requirements between rows of a row-based array [68, 85, 103].

3.4 Supported Operations

Nearly all reviewed approaches support, at most, all logic operations and integer arithmetic, with the frequent exception of integer division. Only some cases support floating or fixed-point arithmetic [39, 67, 73]. Unless the host processor directly controls the accelerator, some comparison operations are also supported to determine when execution should terminate. Mesh arrays favor homogenous units with support for more operations, in some cases resorting to fully fledged *Arithmetic and Logical Unit*s (ALUs) [34, 70], which may contain their own register files [80]. In [5], Bansal *et al.* present a performance analysis of mesh arrays as a function of FUs capabilities. Row designs prefer more specialized units supporting a reduced set of operations, especially if they are tightly integrated [13, 17]. Some designs find a tradeoff, using mostly simple FUs together with a smaller number of more specalized units [67].

3.5 Memory Access Capabilities

Potential speedups are higher for binary segments that contain numerous memory accesses, since there might be large amounts of latent data parallelism. Therefore, support for memory access is required to achieve significant speedups. However, memory access support for accelerators is usually an issue. Ideally, the array should be able to access all the application data with high concurrency and low overhead, which means that a sophisticated memory structure is required. The design of this structure is also affected by whether the accelerator is tightly or loosely coupled.

In the former case, the accelerator circuit is usually restricted to accessing the host processor pipeline or register file. For example, per iteration of the accelerated segment up to one store operation is supported in References [17, 67] by resorting to the host pipeline's writeback stage. In Reference [40] the memory access instructions of the accelerated segments are indirectly supported by executing them through the host. Exceptionally, concurrent load accesses are supported in Reference [8] via multi-port memories, and in Reference [7] a memory controller can handle accesses from address generator units on the accelerator, into either local shared scratchpad or to external memory via a shared cache.

Loosely coupled accelerators can afford a more complex memory architecture design. For example, private local memories [56, 83], with data being moved between them and the main memory before and after execution; other designs resort to a shared memory between the processor and the accelerator [7, 34, 44, 58, 73]. Some implementations share the entire data memory. The implementation presented in Reference [58] relies on a dual-port memory, reserving one port for the processor and another for the accelerator, allowing for one memory access per clock cycle with a known pattern. The scheme presented in Reference [73] supports two concurrent arbitrary accesses per clock cycle by the accelerator. However, neither supports accesses to data residing in external memory.

Other systems share only one or more defined ranges. For instance, in Reference [50] a cache is used to share one runtime defined range, bypassing the need to write accelerator results to main memory. Determining appropriate memory ranges is required, and if data accesses have a small locality, the array may frequently need to access data outside the range. The work in Reference [51] studies the distribution of data through several local single-port memories. Data access is first profiled, and data arrays are interleaved throughout the memories. A compile-time analysis determines an adequate mapping of arrays to local memories, based on array sizes and access frequency. Superfluous memory operations are also removed to reduce pressure. The approach presented in Reference [56] generates a custom multi-port cache architecture per segment to accelerate by analysing the memory accesses performed by the segment and separating them into disjoint regions. The accelerators's caches access other memory components via peripheral bus, and coherency between them and main memory is also addressed. Similarly, in Reference [14] instruction traces are used to generate accelerators with tailored multi-port memories, but the synthesis process works by feeding the memory information retrieved from the traces back into the compiler in the form of *C* code.

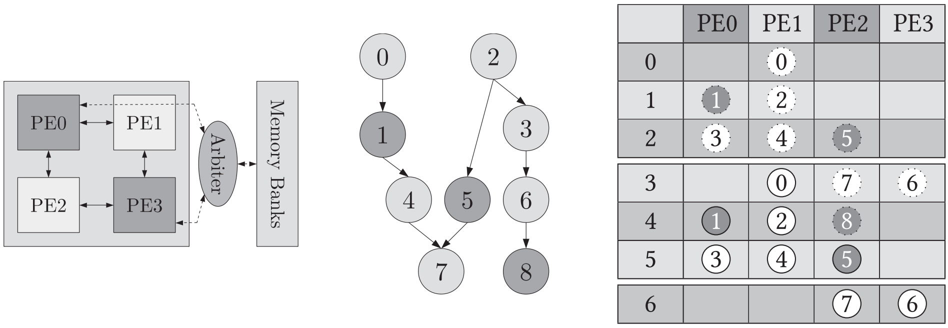
Another issue is the type, and pattern, of the memory accesses supported. That is, read and/or write, and whether support exists for accesses to arbitrary addresses or if it is limited to pre-configured strides. In References [58, 73], the former is possible by having several dedicated load/store FUs capable of using runtime results of other FUs as addresses (including byte-addressing for Reference [73]), and in References [2, 58] address generators are used, with pre-configured start adresses, iteration count and stride. In Reference [65], an address generation method and architecture capable of complex access patterns are proposed for application in heterogeneous systems with shared memories.

3.6 Execution Model

How execution is controlled and carried out on the accelerator depends on the target binary segment and on the specific details of the accelerator architecure. In general, the CDFGs of the target segments are processed such that one or more instructions or configuration words are generated. These instructions are composed by the control bits for the logic of the accelerator, including FU activation, multiplexer switching, and enable bits for any internal register files or local memories. Alternatively, the approach may entail more specialization of the accelerator architecture, generating hardware that requires less control information to operate or none at all. We identify two aspects to the control of the accelerator that come as a consequence of the specific details of the translation process: (1) if the target CDFGs are temporally partitioned (i.e., multiple configuration words are required) and (2) if loop pipelining/modulo-scheduling is supported.

The main reason for partitioning the target CDFG intos sub-graphs is so that fewer resources can be utilized to execute the entirety of the graph. For example, when not resorting to temporal partitioning, accelerators tightly coupled to the host pipeline can be controlled by a single custom instruction [8, 17, 44, 80] through instruction set extension [36]. The translation is simpler for this strategy, but the supported size of the CDFG is limited by resource availability. Instead, by dividing the graph into timesteps, the same accelerator resources can be utilized to execute multiple operations within the graph [67]. To exploit this potential for resource savings, more sophisticated translation is required, and possibly hardware to store and forward data between timesteps.

We find that partitioning as we have just exemplified is applicable to both acyclic and cyclic segments for the same purpose of resource savings or as a consequence of resource limitations. However, the loop pipelining/modulo scheduling aspect is only applicable to cyclic segments. The purpose of this well-known strategy [52, 76] is to increase performance by initiating iteration i+1 during execution of iteration i as early as possible, i.e., with the smallest *Initiation Interval* (II). The



	PE0	PE1	PE2	PE3
0		0		
1	1	2		
2	3	4	5	
3		0	7	6
4	1	2	8	
5	3	4	5	
6			7	6

(a) target mesh architecture (b) target sub-graph (c) modulo reservation table

Fig. 8. Modulo scheduling performed on mesh arrays (adapted from Reference [51]). The dark Processing Elements (PEs) in Figure 8(a) are capable of memory accesses, and dark nodes in Figure 8(b) are memory access operations. In Figure 8(c), dashed nodes represent iteration *i-1*, and solid nodes represent iteration *i*.

II is determined either by the backward connections within the segment CDFGs, i.e., data/control dependencies between iterations, or by resource availability, which if insufficient delays execution of operations to later timesteps. When targeting row-based arrays, the most straightforward way to implement loop pipelining is to instantiate one FU per CDFG node, to connect the units according to the CDFG edges, and to generate the control bits for register and FU enables at the correct times [72]. This approach does not scale well for very large CDFGs, and we have found no row-based designs that simultaneously partition the graph, while overlaping execution of multiple iterations. Additionally, it is hard to exploit resource reutilization between accelerated segments. In contrast, mesh designs perform better by virtue of the richer and omnidirectional interconnects.

An example of loop pipelining via modulo scheduling [76] on meshes is shown in Figure 8. Modulo scheduling is a widely used algorithm for instruction scheduling, traditionally used for software pipelining. In Figure 8(c) the nodes of the CDFG shown Figure 8(b) are scheduled on the target array presented in Figure 8(a). Rows of the table represent timesteps, and columns represent FUs. A total of seven timesteps is required to schedule a single iteration of the entire CDFG, but by repeating the scheduling pattern (represented by rows three to five), an iteration can be initiated, and therefore completed, at every three timesteps. By concurrently executing operations belonging to different iterations, greater speedups and better resource utilization per cycle are possible.

3.7 Summary of Accelerator Architecture Taxonomy

Table 2 shows the summarized features and values for the accelerator architecture taxonomy. For the binary translation aspect of the approaches, we considered that favouring online implementations was preferable to interfering with compile and design flows. Our rationale for classifying the architecture features is similar, although there is no universal criteria adequate for all features.

Regarding the interface, we sort the possible implementations from most intrusive, to least intrusive, i.e., more transparent. Transparency is lowest for approaches where the coupling is tight, and modification to the application binaries is required and is highest for loosely coupled approaches where standard interfaces are used to exchange data and few/no modifications to the host processor architecture or its instruction set are required.

For arrangement, we sort from generic layouts to specialized designs, and as for interconnections, richer connection capability facilitates runtime translation. We consider connectivity poorest when connections between FUs and other elements are completly specialized and non-configurable and is richest if full connectivity is implemented. Regarding FU operations, more

Table 2. Summary of Accelerator Architecture Taxonomy Features and Values

Feature	Key	Value #1	Value #2	Value #3	Value #4
Interface	A	Least Intrusive	-	-	Most Intrusive
Arrangement	B	Mesh	Multiple Rows	Single-Row	Custom
Interconnect	C	Poorer	-	-	Richer
Operations	D	Some Int./Logic	All Int.	*D2* + Fixed Pt./Some Float Pt.	*D2* + Float Pt.
Mem. Access	E	None	1 Sequential	1 Arbitrary/>1 Sequential	>1 Arbitrary
Exec. Model	F	Least Complex	-	-	Most Complex

and larger binary segments can be targeted by supporting the largest possible set of operations, and likewise for memory accesses, where more accesses per clock cycle increases potential for acceleration.

Finally, we classify the execution model based on a qualitative perception of complexity, from least to most complex as follows: (1) a single, pre-execution, configuration for the accelerator, without support for loop pipelining/modulo-scheduling; (2) multiple configuration words fed to the accelerator during execution, without support for loop pipelining/modulo-scheduling; (3) a single, pre-execution, configuration is fed, with support for loop pipelining/modulo-scheduling; and (4) multiple configuration words fed to the accelerator, with support for loop pipelining/modulo-scheduling.

The discussion in the following section will classify the accelerator architecture of each approach according to these values (e.g., *A4, B1, C3, D3, E3, F2*).

4 STATE-OF-THE-ART APPROACHES TO BINARY ACCELERATION

The following sections summarize notable state-of-the-art approaches to designing systems capable of autonomous runtime acceleration. We note that readers intending to know more about additonal research aspects of the topics involved can access the following surveys. Binary rewriting is surveyed extensievly in Reference [97]. A general view of hardware/software co-design is presented in Reference [99], an extensive look at several aspects of reconfigurable architectures is shown in Reference [20], and References [15, 45] are surveys of mesh architectures, some of which are also presented here. Table 3 summarizes architectural and methodological information for the approaches detailed here, and Figure 16 and 17 present qualitative categorizations of both aspects.

4.1 Warp Processor

The Warp processor approach is an FPGA-based runtime reconfigurable system based on binary decompilation [92]. The proposed system is composed of a single main processor, a binary profiler, and an on-chip circuit synthesis module, which targets an FPGA device with whom the processor shares data memory. *Basic Blocks* (BBs) are first detected during execution [38]. Once profiled, the binary is decompiled into an intermediate representation that is mapped into a custom FPGA fabric by tools running on an additional processor. The fabric is modeled with an interconnect structure and resource layout amenable to runtime circuit synthesis. After the hardware is synthesized, execution is migrated by modifying the program binary, and operands are fetched from memory.

Decompilation is performed so that high-level information can be retrieved from the executing binary. This process includes state-of-the-art techniques (e.g., reconstructing *if-else* statements) and two proposed techniques: *loop re-rolling* and *operator strength promotion*, which recovers operations that the compiler replaced with lengthier equivalent sequences. When comparing the speedups achieved by generating circuits with the decompiled binary, versus directly using *C* code

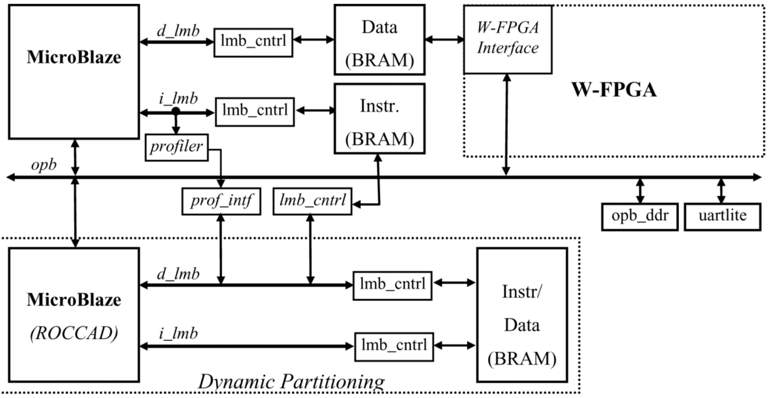


Fig. 9. The MicroBlaze-based Warp processor system. An additional processor performs runtime binary segment detection and translation [58].

equivalents, the former is only 2.5% less efficient. However, without resorting to the two proposed decompilation techniques, this increases to 33%. Without resorting to binary decompilation, the authors claim that speedups versus software execution would not be possible.

Circuit synthesis onto the custom fabric is 25× faster than commercial tools running on a desktop workstation targeting conventional FPGAs, requiring 3.6 MB of memory to execute on a 40 MHz ARM7. The FPGA's functionality was verified via simulation targeting a 0.13 μm technology. Using an ARM9 as the main processor operating at 200 MHz and the custom FPGA at 100 MHz, an average arithmetic speedup of 6.5× was achieved for a set of single-threaded applications that do not contain floating-point arithmetic, recursion, pointers, or dynamic memory allocation.

This approach is evaluated on commercial FPGAs, using the MicroBlaze softcore as the main processor, and is also extended to target multi-processor systems [58]. As shown in Figure 9, the profiler observes the instruction stream and dispatches the most frequently executing segments to an auxiliary MicroBlaze-based subsystem, which performs the on-chip synthesis. This processor is instantiated without data caches or a floating-point unit (to reduce area overhead), requiring 11 s on average to generate a circuit for the custom fabric. If synthesis of a segment fails due to insufficient resources or a long synthesis time, then execution simply falls back to software.

The fabric was validated by simulation targeting 0.18 μm technology. The full system was tested on FPGAs by replacing the proposed fabric with custom hardware implementations of the targeted segments. Compared to a single MicroBlaze processor operating at 100 MHz, an average speedup of 5.1× is achieved for six integer benchmarks from multiple suites [60, 90], and energy consumption is decreased by 65%, when the fabric and all other components operate at 200 MHz.

Binary Translation – Segment Type *A1*, Detection *B3*, Translation *C3*, Binary Mod. *D3*, Acc. Type *E3*

Acc. Architecture – Interface *A4*, Arrangement *B4*, Interconnects *C3*, Ops. *D2*, Mem. Access *E2*, Exec. Model *F3*

4.2 AMBER/ADEXOR

A processor system named *Adaptive Extensible Processor* (AMBER) capable of augmenting its instruction set post-compilation is presented in Reference [68]. The generated custom instructions execute on a heterogeneous accelerator, first introduced in Reference [62], which is tightly coupled to the processor pipeline. The host processor is a four-issue in-order RISC supporting the

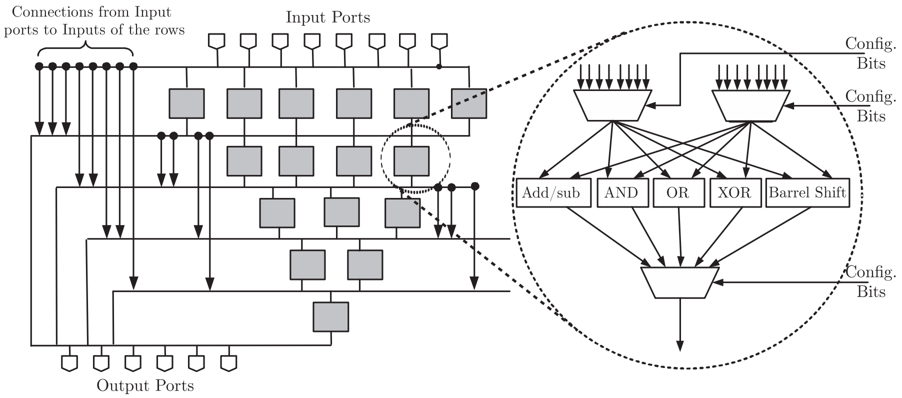


Fig. 10. FU array coupled to the processor pipeline in the AMBER/ADEXOR approach (adapted from Reference [68]).

MIPS instruction set. The accelerator, shown in Figure 10, is composed of FUs that support all the instructions of the host instruction set, except for loads, division, and multiplication. Only one store operation is supported per configuration. The FUs are organized in rows, with data flowing to subsequent rows.

The approach targets hot BBs, expressed as sub-graphs, which are first detected offline via simulation, then mapped to the accelerator, and, finally, executed via a custom instruction fed to the processor. Each custom instruction has its own latency, depending on the depth of the CDFG, and represents a binary segment with a single entry, which does not cross loop boundaries. Some BBs are too large to map to the resulting architecture in a single custom instruction (i.e., accelerator configuration), in which case they are partitioned and implemented as a sequence of custom instructions. The details of the accelerator design were determined via a quantitative approach based on analysing BBs from 19 kernels, which determined aspects like number of inputs and outputs and number of rows. An instance containing 16 FUs and eight inputs and six outputs (see Figure 10) was found to be sufficient to support nearly all tested BBs. With these parameters, 512 bit are required to configure the accelerator per custom instruction. The final design was synthesized for a target technology of 0.18 μm, and the resulting occupied area was approximately half than that of a 32 kB cache implemented in the same technology. For 14 benchmarks from the MiBench Suite [69], an average speedup of approximately 1.2× was achieved when coupling the accelerator to a MIPS-based host processor operating at 300 MHz.

The system is extended to support custom instructions with multiple-exits in Reference [67]. Sequences of forward-jumping BBs containing at most four exit points are supported. The sequences cannot cross function returns, indirect branches, or branch and link operations. Invalid instructions (e.g., division) are also considered exit points. When constructing a sequence of BBs, if execution is equally frequent through both the taken and non-taken directions, then both paths are included into the sequence. Only short forward jumps are considered, as long jumps would require support for executing a large number of instructions on the accelerator. If one execution path is heavily biased, then only the former is included in the trace. Supporting multiple exits increases the average number of instructions in the accelerated traces from the 6.39 achieved in Reference [68] to 7.85.

The traces are then transformed into configurations for the accelerator, which is invoked via custom instructions added to the binary in a post-compilation phase. Alternatively, it is possible to observe the instruction stream at runtime and trigger accelerator configuration and invocation

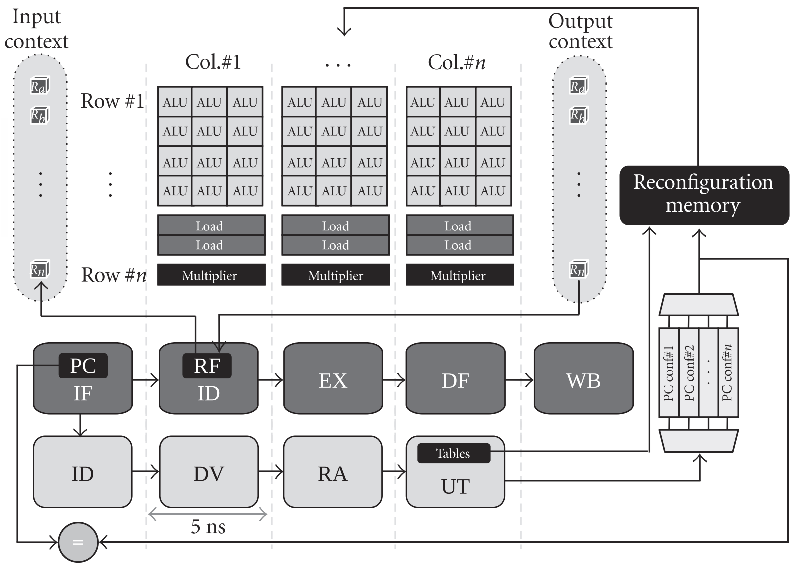


Fig. 11. DIM approach: runtime binary translation for a tightly integrated array (adapted from Reference [80]).

based on the observed program counter. The additional conditional execution hardware increases the average width of the accelerator configuration word to 615 bit, but the total amount of configuration memory required is reduced by compression via a similarity analysis on the generated configurations. Up to 32 custom instructions are supported. The accelerator is composed entirely of combinatorial logic, meaning that outputs are not registered at each row. This means that the latency of the accelerator varies per configuration (based on the depth of the mapped CDFG). The exact latency from the point of view of the host processor depends on its own frequency.

Given this, three accelerator variants were tested: two for which the latency is variable as described and a third where latency is enforced at two clock cycles. The former two cases vary between them in number of input/output ports to the processor register file. Considering a base single-issue MIPS processor operating at approximately 130 MHz, and 16 benchmarks from the MiBench suite [69], an average speedup of 1.9× was achieved for the variant with variable accelerator latency and the most input/output ports, which is itself 1.2× faster than the variant with fixed latency. Augmenting the base processor with the accelerator and its configuration memory increases the area requirement by 25% for a target technology of 0.18 μm. The average power consumption of the base processor is 71.5 mW, while the accelerator consumes 229.7 mW. However, the execution time reduction leads to energy savings of 21% in the best case.

Binary Translation – Segment Type *A2*, Detection *B2*, Translation *C2*, Binary Mod. *D2*, Acc. Type *E1*

Acc. Architecture – Interface *A2*, Arrangement *B2*, Interconnects *C4*, Ops. *D3*, Mem. Access *E2*, Exec. Model *F1*

4.3 *Dynamic Instruction Merging* (DIM)

In the DIM method, presented in Reference [8], configurations are dynamically generated for an array tightly coupled to a MIPS processor, as shown in Figure 11. The array contains homogeneous rows of heterogeneous FUs, including integer arithmetic, logic, and memory access units. Division and floating-point operations are not supported. There are as many memory units as available to the enveloping system architecture, and the array supports cache misses by stalling, but implementation details are not given. Operands are fetched from the processors register file and directed to any FU on any row of the array as needed; connectivity is very rich, with any FU being able to connect to any other. Additionally, it is possible to feed immediate values as inputs to the FUs, which are stored as part of the configuration. The FUs may have different latencies based on their type, and the array does not register the results at each row, meaning that it behaves as a multi-cycle unit, whose exact latency depends on the present configuration. Configurations are generated by observing the instruction stream and detecting sequences delimited by either branches or unsupported operations. Hazard analysis and speculative execution of up to three BBs are also supported.

For experimental evaluation, different variants of the array were tested. For each variant, three different speculation policies were used, as well as different cache sizes for the configurations. For 18 integer benchmarks of the MiBench suite [69], an average speedup of 2.7× was achieved for the best case scenario versus a MIPS R10000 processor, along with a 50% decrease in energy consumption. For a 0.18 μm technology, the smallest instance of the system requires over 600 thousand logic gates, and each array configuration requires over 3,000 bit to encode.

An analytical model is detailed in Reference [80], which demonstrates the need for parallelism exploitation at several granularity levels. To implement this functionality, an evaluation is performed on attaching DIM arrays to multiple SparcV8 processor cores. The DIM arrays do not introduce a critical path in the SparcV8, despite the array being combinatorial. The capability of power gating the array is also introduced; whether an FU is active is part of the dynamically generated configurations. For validation, six thread-based benchmarks, most from the PARSEC [9] and SPLASH2 [100] suites, were used. The benchmarks are first ran on standalone SparcV8 cores, varying the number from 4 to 64, and then again with two variants of the DIM arrays attached. For 4, 8, and 16 cores, ILP exploitation achieves an average speedup of 1.7×; for 64 cores, the average is 1.1×. It is also shown that the performance increase attained by doubling the number of SparcV8 cores from 4 to 8, or from 8 to 16, is comparable to, instead, resorting to DIM arrays for each respective case, when using the smaller array variant. There is only marginal performance increase relative to this scenario by using the larger array instead, and slowdowns in two cases due to longer translation times and configuration cache misses. The DIM array is used in Reference [79] to create a similar system where several instances of this SparcV8-based architecture are grouped into a mesh along with a shared cache. The resulting multiprocessor system is evaluated against two non-accelerated multiprocessor systems: one based on the single-issue SparcV8 and another based on a 4-issue out-of-order SparcV8.

This work is continued in Reference [30], where compatibility for other ISAs is addressed. An additional binary translation layer extends the applicability of the DIM array to binary code compiled for other architectures, e.g., x86, ARM, or Sparc, by translating it to the MIPS binary code supported by the existing binary translation that targets the array. Both translation steps are implemented in hardware. For the implementation presented, only x86 code is addressed as a proof-of-concept. A total 50 integer instructions can be translated to MIPS of the 190 present in the targeted x86 IA32 instruction set. Support for interrupts and extensions (e.g., MMX and SSE) is left for future work. It is important to note that extensions were added to the MIPS instruction set in this implementation to minimize the translation overhead, i.e., to reduce the number of MIPS instructions required to implement an equivalent x86 instruction. The system was evaluated via simulation targeting a 90 nm library, employing a MIPS R3000 processor as the host. An additional instruction set simulator was used to retrieve x86 traces. Eight benchmarks from the MiBench suite [69] were used. The average speedup when executing translated x86 code on the extended MIPS processor coupled to the accelerator is 1.5×, with a considerable impact being introduced due to the x86 to

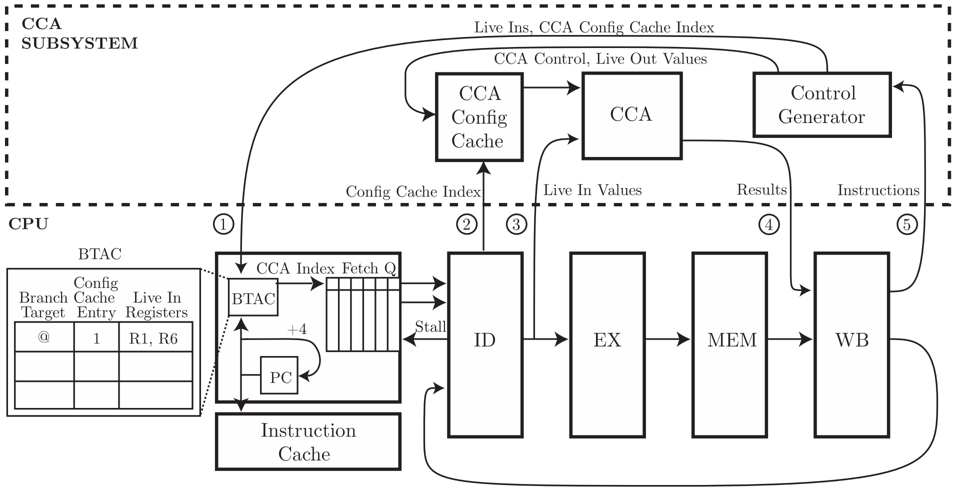


Fig. 12. The CCA array is coupled into the processor pipeline. Execution is shifted toward the CCA after generating configurations for offline delimited regions of code [17].

MIPS translation. Executing the translated code on the MIPS processor without resorting to the accelerator results in a slowdown of 2.0×, relative to executing native MIPS code.

Binary Translation – Segment Type *A2*, Detection *B3*, Translation *C3*, Binary Mod. *D3*, Acc. Type *E1*

Acc. Architecture – Interface *A2*, Arrangement *B2*, Interconnects *C4*, Ops. *D2*, Mem. Access *E4*, Exec. Model *F1*

4.4 ***Configurable Compute Accelerator*** **(CCA)**

A method for transparent instruction set extension is presented in Reference [19]. An ARM pipeline is augmented with a CCA as shown in Figure 12. The CCA is single-cycle unit with a row-based design with inter-row crossbars (see Figure 7(b)). A quantitative analysis of 29 applications determined the number of rows, columns, and type of units. The proposed CCA has four inputs and two outputs, and several combinations of depth and widths were tested. Each row contains only one type of FU that supports either arithmetic and logical operations, or only the latter. Memory, barrel-shift, multiplication, division, and branch operations are not supported.

Segment detection is based on traces, to extract sequences of BBs where the *branch* operations of the blocks are highly biased. This can be performed online by observing a trace cache, or offline at compile-time through an intermediate simulation step. The latter introduces special binary instructions that delimit the code to translate at runtime. Evaluation was performed using the SimpleScalar simulator with 29 benchmarks. A CCA of depth 4 was shown to be able to execute 82 % of the candidate graphs and achieved an average speedup of 1.3× for 11 benchmarks versus an ARM 4-issue processor [19]. In Reference [17] an exploration of domain-specific CCA designs was performed, together with relying only on compile-time segment detection to reduce runtime overhead, and a CCA configuration cache to store previously translated segments. For a set of 22 benchmarks, the average speedup over a baseline ARM-926EJ was 2.1×.

The CCA is also employed in Reference [18] to address binary portability for hardware accelerated systems. Applications developed to use a given accelerator might not be compatible with future hardware implementations. To address this, a virtualization module is introduced that monitors the instruction stream and generates configurations and control for the accelerator present

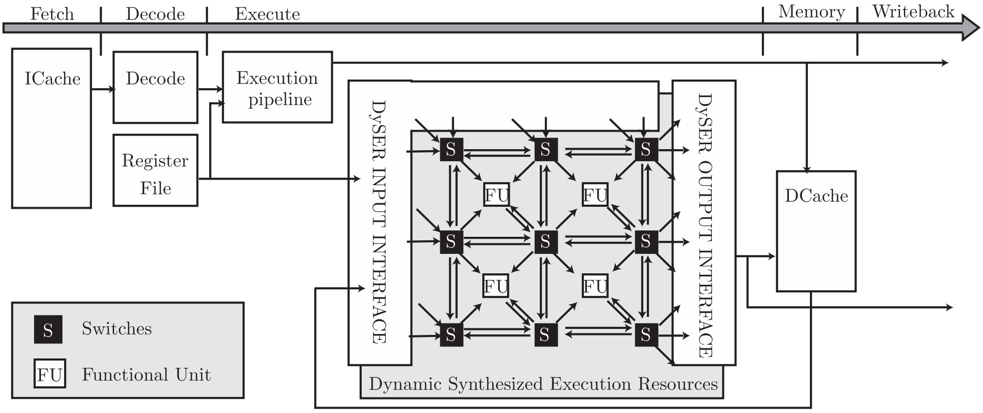


Fig. 13. DySER Architecture and Organization (adapted from Reference [40]).

in the system. To validate this, a loop accelerator architecture that executes modulo-scheduled loops is employed. The accelerator integrates the CCA as a functional unit. It was designed based on profiling data by determining the required resources to achieve the best possible speedup for the candidate loops. The accelerator supports integer, single- and double-precision floating-point operations as well as 16 address generators for load operations and is composed of reconfigurable heterogeneous FUs. Scenarios in which one or more steps of the binary translation of loops are either performed online (by the virtual machine) or offline (through static compilation steps) are analyzed. An average speedup of 2.7× is achieved for 38 benchmarks versus a single-issue ARM, using the chosen hybrid combination of online/offline binary translation without compromising binary portability.

Binary Translation – Segment Type *A2*, Detection *B2*, Translation *C3*, Binary Mod. *D1*, Acc. Type *E2*

Acc. Architecture – Interface *A2*, Arrangement *B2*, Interconnects *C3*, Ops. *D1*, Mem. Access *E1*, Exec. Model *F1*

4.5 *Dynamically Specialized Execution* (DySE)

The DySE approach is based on identifying path-trees of basic-blocks at compile-time [40] and executing frequent paths on the proposed, tightly integrated, array composed by FUs, switchboxes, and other elements, collectively named *Dynamically Specialized Execution Resource* (DySER). The target segments are detected by constructing a tree containing all possible paths through a set of BBs comprising a trace loop and then extracting the most frequent paths. The paths are then split into memory handling instructions (load-slice) and computation instructions (computation-slice). The former are executed on the processor side, and the latter by the DySER array. Extensions to the instruction set are used to implement communication between the processor and the array, as well as to allow for use of the processor as a mechanism to ensure that load-order is maintained.

The FUs in the array are heterogenous and connected to their four nearest switchboxes, as shown in Figure 13. The FUs are multi-cycle units that resort to handshaking signals to indicate the presence of valid data at their output. The operations supported by the FUs were chosen based on a quantitative analysis and include integer and floating-point arithmetic and logic operations. Specific instructions are added by the compiler to the binary so the processor can configure the array and exchange data. Configuration happens prior to execution by propagating the control information through the FUs themselves. This avoids the use of an additional configuration memory but imposes a longer configuration time and requires generating the processor instructions for this purpose. Additionally, multiple DySER blocks can be coupled to the processor. Since control of the array is performed via software through custom binary, it is possible to interweave execution on one DySER block with simultaneous configuration of a second block resorting to a prefetching scheme, mitigating configuration overheads. Similarly, it is possible for execution to be pipelined on a single array by software-pipelining of the load-slice that executes on the processor.

The area and power requirements were evaluated for a 55 nm technology. In the evaluated design, a third of the blocks are floating-point units, and the remainder are integer ALUs. A DySER block with 64 FUs requires the same area as a single-port 64 kB RAM, and at an operating frequency of 2 GHz consumes 1.9 W. Evaluation was performed via a cycle-accurate simulator by coupling DySER arrays to single-, dual- and four-issue out-of-order processors. The arrays supported up to eight concurrent invocations via pipelining. For benchmarks from the PARSEC [9], SPEC CPU2006 [21], and Parboil [41] suites, the compiler stage shows that over 90% of an application can be covered by accelerating only the five most frequent paths per detected path tree, where the number of instructions per path ranges between 50 and 400. Resorting to two blocks with 64 FUs, up to 70% of an application can be mapped, achieving a geometric mean speedup of 2.1× and energy reduction of 40% (or 60% if no speedup is required), when coupled to a dual-issue out-of-order processor. For all cases, the bootleneck is due to the processor driven load-slice.

The DySER approach is enhanced in Reference [39] with translation techniques and architecture improvements that allow for further exploitation of data workload parallelism, despite the heterogeneous nature of DySER's FUs. The authors present three major types of graph transformations. To avoid underutilization of DySE resources, loop unrolling and strip mining can be used. The first simply replicates the nodes of the graph while reducing loop count, while the second is a transformation aiming to vectorize the graph by coalescing memory accesses into vectorized loads/stores. However, over-sized graphs can be reduced by employing a sub-graph matching approach, which decomposes the graph into sub-configurations of the array. This last transformation, along with vectorization, require additional hardware on the array. To support quick configuration changes to execute sequences of sub-graphs, each FU contains a local configuration memory, and to support vector memory operations, wide memory ports are required as well as logic to map data vectors to the array's input ports.

Binary Translation – Segment Type *A3*, Detection *B2*, Translation *C1*, Binary Mod. *D1*, Acc. Type *E1*

Acc. Architecture – Interface *A1*, Arrangement *B1*, Interconnects *C2*, Ops. *D4*, Mem. Access *E1*, Exec. Model *F4*

4.6 BERET

The BERET approach is presented in Reference [44]. It aims to minimize power used in the fetch and decode stages by accelerating sub-graphs, thus avoiding conventional reads/writes of both intermediate values and instructions. In BERET, a compiler-driven flow detects Superblocks [46], i.e., frequent BB sequences detected via profiling. Only the subset where last BB has an 80% likelihood of branching to the first is considered. The CDFG representing the loop trace is split into sub-graphs that the accelerator executes as atomic units. BERET's compilation framework is based on Trimaran [91].

The accelerator itself is tightly coupled to a host ARM pipeline, between the issue and execute stages and has access to the ARM's register file, as well as its L1 data and instruction caches. BERET is composed of several *Sub-graph Execution Block*s (SEBs), as shown in Figure 14. The blocks contain different heterogeneous sets of units (ALUs, multipliers, shifters, memory access ports, as well as interconnection logic). The eight different blocks were designed based on a quantitative analysis

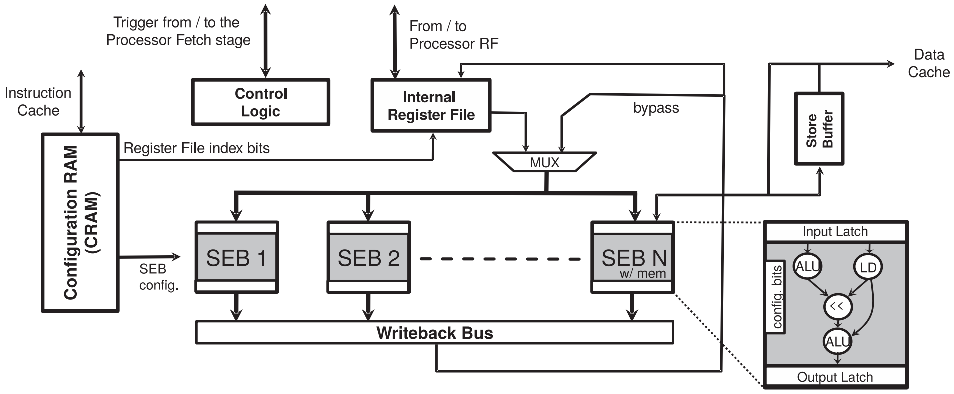


Fig. 14. *Sub-graph Execution Blocks* (SEBs) in BERET coupled to processor pipeline (adapted from Reference [44]).

of a trained set of applications. Each sub-graph of a detected trace is scheduled onto the smallest SEB that is capable of executing it (to maximize hardware usage). Data are transported between blocks via an internal register file. At runtime, BERET is invoked when execution reaches any of the addresses annotated by the compiler. This causes BERET to retrieve configuration bits, which were added to the program binary by the compiler via the shared L1 cache. The host processor is stalled while BERET executes the sequence of sub-graphs that compose the trace. Although each SEB exploits the ILP in the sub-graphs mapped to it, no loop pipelining is exploited. When the trace CDFG is partitioned for scheduling, each resulting sub-graph has only one exit condition (i.e., branch instruction); execution ends when any of the subgraphs triggers its exit condition.

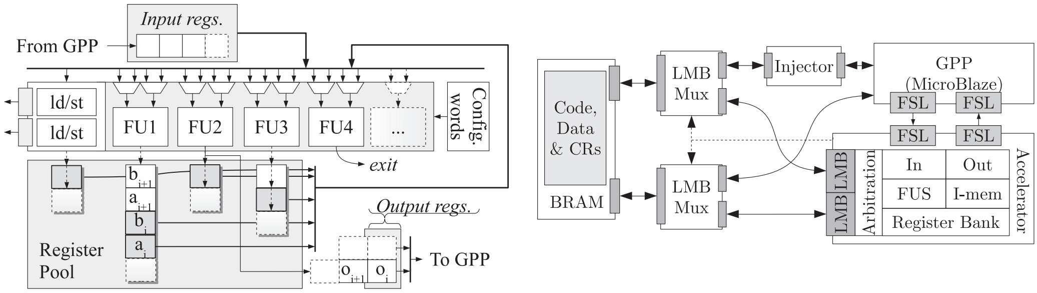
The execution blocks of BERET were designed via an experimental approach based on a set of 12 benchmarks from the SPEC2000, SPEC2006, and MediaBench benchmark suites [21, 54]. The hot traces of each benchmark were extracted (ranging from 2 to 50) and used to define the set of SEBs that would allow for all hot traces to be mapped, while also maximizing the size of the sub-graphs. The average trace contains 20 instruction and operates on up to 6 register file inputs/outputs; the resulting SEBs execute sub-graphs with an average of 2 to 6 instructions each. An additional set of 12 benchmarks is used to validate the approach. The host processor is an ARM1176, a single-issue in-order processor operating at 800 MHz. The authors estimated the execution time for the ARM and for BERET in cycle-accurate simulators. For power and area estimations, the design was synthesized targeting 65 nm, and a clock frequency of 800 MHz. Execution time was reduced by an average of 10% due to ILP exploitation. Energy consumption for the entire application is reduced by 35% on average; when considering only the accelerated traces, 3× less energy is required. The area required by BERET is 5× smaller than the ARM, occupying 0.4mm^2.

Binary Translation – Segment Type *A3*, Detection *B2*, Translation *C1*, Binary Mod. *D1*, Acc. Type *E1*

Acc. Architecture – Interface *A3*, Arrangement *B3*, Interconnects *C3*, Ops. *D2*, Mem. Access *E1*, Exec. Model *F3*

4.7 *Custom Loop Accelerator* (CLA)

An approach that automatically generates a CLA from runtime traces is presented in References [72, 73]. The accelerator is transparently used at runtime as a loosely coupled co-processor by the host MicroBlaze processor. To generate a custom accelerator instance, the execution trace is first analysed by executing the target binary in a cycle-accurate simulator. This analysis detects



(a) CLA architecture

(b) System architecture

Fig. 15. System and accelerator architecture for the *Custom Loop Accelerator* (CLA) approach [73].

frequently repeating patterns of instructions, creating single-entry multiple-exit sequences called Megablocks [10], which are used to generate a tailored accelerator instance.

The coarse-grained template shown in Figure 15(a) is tailored such that it contains only FUs necessary to schedule each loop (i.e., Megablock) at its minimum possible II [73] . The only type of operation that may increase the II are memory accesses. Every CLA instance contains two memory ports, which directly connect to the MicroBlaze's data memory, as seen in Figure 15(b). Supported operations include all integer and single-precision floating-point arithmetic, including *float-to-int* conversion and vice versa. Most FUs are single-cycle, while some of the floating-point units require three to four cycles. All units are fully pipelined, meaning that the executed instructions are not atomic, i.e., instructions keep being executed despite activation of multi-cycle units. The CLA implements only the minimum required multiplexer connectivity required to implement all the connections between scheduled operations. As more loops are scheduled during the generation process, both FUs and existing connections are reused whenever possible. The scheduling aims for the lowest II possible, meaning that despite resource utilization between supported loops, minimum resource usage is not guaranteed, since determining a specific scheduling order and II for each loop could reduce the number of units and multiplexers further. The accelerator is invoked at runtime by observing the instruction address bus and introducing a branch to a communication routine that sends operands from the MicroBlaze's register file, stalls the processor during accelerator execution, and, finally, retreives results. Finally, the subroutine branches back to the address where execution was interrupted. Accelerating two Megablocks with the same start address is unsupported.

The approach was evaluated in Reference [73] with integer and floating-point kernels from the Texas' IMGLIB library [47] and the Livermore Loops set [75], versus a MicroBlaze-only baseline. The testbed was a Xilinx VC707 Evaluation Board, with all modules in the system operating at 110 MHz, including the CLA. The average Megablock contains 34 instructions, which the CLA executes at an average of 5.2 instructions per clock cycle, achieving a mean geometric speedup of 5.6× versus the MicroBlaze. The average CLA requires 1.1× the *Lookup Table*s (LUTs) and 1.8× the *Flip Flop*s (FFs) the MicroBlaze requires. Although all instances of the CLA operated at 110 MHz, the cases without floating-point units achieve an average synthesis clock frequency of 290 MHz.

In Reference [74], the CLA is augmented with *Dynamic Partial Reconfiguration* (DPR) to minimize the area used and avoid decreases in operating frequency when targeting many loops. A single partially reconfigurable region is outlined, containing all FUs, multiplexers, and register pool. The memory access ports remain in the static region of the accelerator. Each variant generated for the reconfigurable region can support any subset of the full set of CDFGs to support. The potential for area savings is evaluated by generating CLA instances without partial reconfiguration support as per Reference [73] and equivalent instances where each loop is implemented as a circuit variant. Area reductions upward of 1.6× can be achieved for CLAs supporting between four to six loops.

Binary Translation – Segment Type *A3*, Detection *B2*, Translation *C2*, Binary Mod. *D2*, Acc. Type *E2*

Acc. Architecture – Interface *A4*, Arrangement *B3*, Interconnects *C2*, Ops. *D4*, Mem. Access *E4*, Exec. Model *F4*

4.8 Other Approaches

The previous section has presented full-fledged representative approaches, each comprising its own body of work. The general methodology is based on either offline or online profiling and binary translation, targeting tightly coupled sub-graph accelerators or loosely coupled loop accelerators. This section summarizes other work efforts that, while related, divert their focus to other details like the scheduling mechanism or memory architecture, or present customized standalone architectures, while still focusing on generating custom hardware for the compute intensive tasks in applications.

4.8.1 Programmable Loop Accelerator. Fan et al. [31, 32] present a constraint-based formulation used to implement a modulo scheduler that generates a custom instance of a loop accelerator. The accelerator is composed of a single row of FUs with local shift registers to hold results. The specific single-operation FUs and interconnections between them are tailored to the target loop. Execution is controled by a finite-state machine that implements all the steps of the generated modulo schedule. The approach then aims to address the lack of programmability of a dedicated architecture by introducing architectural generalizations, such as enchancing the FUs with more operations, additional interconnections, bigger shift registers, and replacing the finite-state machine with an instruction memory. The efficiency of the generalizations is studied by iteratively modifying a CDFG and attempting to schedule it onto the accelerator generated for the unmodifyed version of the same CDFG. The described *Programmable Loop Accelerator* (PLA) is evaluated in terms of benefits and costs of the additional programmibility for 10 loop kernels containing from 17 to 60 operations [31]. Each generated PLA was compared to its single-loop equivalent, as well as with a five-stage in-order RISC processor. All designs targeted 0.13 μm technology, with the loop accelerators being synthesized for 200 MHz and the RISC baseline for 300 MHz. The power consumption of the PLAs is between 2× to 9× that of their single-loop counterparts. This is comparable to the RISC processor, while achieving speedups over it ranging from 6× to 33×. The PLAs require twice the area than the single-loop accelerators and half the area of the RISC processor.

Binary Translation – Segment Type *A3*, Detection *B1*, Translation *C1*, Binary Mod. *D1*, Acc. Type *E2*

Acc. Architecture – Interface *A3*, Arrangement *B3*, Interconnects *C2*, Ops. *D2*, Mem. Access *E3*, Exec. Model *F4*

4.8.2 PolyMorphic Pipeline Array. A multi-core processor named *PolyMorphic Pipeline Array* (PPA) is presented in Reference [71]. It is composed of a mesh of homogeneous cores, each containing an instruction memory, and four FUs with individual 16-entry register files. The approach focuses on providing both coarse- and fine-grained parallelism by exploiting outer and inner loop pipelining. Each core fetches its own instruction from local memories and also contains a local register file. The FUs support integer arithmetic, with only one unit per core supporting multiplication. All processing elements in the same column share access to a part of a global set of data memories via a shared bus. The memory access latency varies depending on how port sharing is configured. Cores are connected to their nearest neighbours but can also send/receive instructions to/from other cores through a virtualization mechanism that allows the PPA to choose the number of cores used to execute loops, which are modulo-scheduled at complile-time, depending on available resources. The performance driven from modulo-scheduling inner loops is evaluated using three applications. Over executing the entire application on a single PPA core, resorting to modulo-scheduling the inner loops achieves an average speedup of 2.5×. Additionally, the speedup is increased by 1.7× by allowing for dynamic re-scheduling of the loops. A PPA with eight cores operating at 200 MHz synthesized for a 90 nm technology requires approximately 1.4× the power of an ARM11, but the respective speedup allows for an energy consumption reduction of 8.3×.

Binary Translation – Segment Type *A1*, Detection *B1*, Translation *C1*, Binary Mod. *D1*, Acc. Type *E1*

Acc. Architecture – Interface *A4*, Arrangement *B1*, Interconnects *C2*, Ops. *D2*, Mem. Access *E4*, Exec. Model *F4*

4.8.3 Paek et al. In Reference [70] a framework is presented targeting a loosely coupled mesh by disassembling compiled binary and localizing static loops and their iteration counts. The approach discards any loops that have an unknown or non-constant iteration count, and targets those that iterate over a given threshold. Inner loops are unrolled if their iteration counts are also constant. The resulting CDFGs are processed by a constraint-aware scheduler to generate configurations for the array. To minimize the overhead due to data transfers between the processors main memory and the array's scratchpad, the translation modifies the program binary. The array is composed of homogeneous processing elements, containing basic ALUs that are connected either to their nearest neighbours, or globally, based on the chosen strategy. Two operation modes are supported: Either all elements fetch their own configuration per stage from a local context memory or the leftmost column of elements accesses the configuration memory and then propagates it to the following columns, effectively implementing loop pipelining. The array supports sequential accesses to memory, requiring that data be sorted in the scratchpad. To achieve this, memory addresses of write operations, performed on data the array will access, are modified at the CDFG level to target the scratchpad. If this optimization is impossible (e.g., due to complex pointer arithmetic), then the approach falls back onto explicit data copying. An ARM7 processor was used as a host for an 8 × 8 array with nearest-neighbor connections, achieving an average speedup of 9.4× for selected kernels from the DSPstone suite [105]. For a JPEG decoder application, where several kernels were accelerated, a speedup of 1.5× was achieved.

Binary Translation – Segment Type *A1*, Detection *B2*, Translation *C2*, Binary Mod. *D2*, Acc. Type *E1*

Acc. Architecture – Interface *A3*, Arrangement *B1*, Interconnects *C3*, Ops. *D1*, Mem. Access *E2*, Exec. Model *F4*

4.8.4 A Just-in-Time Customizable Processor. Chen et al. [13] present the *JIT Customizable* (JiTC) processor, which is based on a multistage *Specialized Functional Unit* (SFU). The SFU can be reconfigured in a per-cycle basis and was designed after analysis of 21 applications. It is tightly coupled to the processor pipeline, has four inputs and two outputs from and to the register file, and contains three heterogeneous FUs. These units support logic and arithmetic operations and include joint ALU-shift and multiply-add operations, but memory and control flow operations are not supported. Common instruction sequences are detected at compile time and transformed into custom instructions targeting the SFU. Custom instructions are executed on the SFU in a single clock cycle, and nearly 90 % of all CDFGs of the custom instructions have a critical path (i.e., depth) of three operations. For CDFGs of higher depth, graph partitioning can generate multiple custom instructions to allow for their execution on the SFU. Nearly all custom instructions execute in under four clock cycles. Evaluation on a cycle accurate simulator demonstrates that average speedups of 1.2× and 1.2× are achieved for the profiling benchmarks versus in-order and out-of-order ARM processors, respectively. For the out-of-order pipeline, four SFUs were used. For 14 other benchmarks, JiTC achieved average speedups of 1.1×. When synthesizing for a 45 nm technology, the SFU can operate at 600 MHz.

Binary Translation – Segment Type *A1*, Detection *B1*, Translation *C1*, Binary Mod. *D1*, Acc. Type *E1*

Acc. Architecture – Interface *A2*, Arrangement *B2*, Interconnects *C2*, Ops. *D2*, Mem. Access *E1*, Exec. Model *F1*

4.8.5 Ferreira et al. An approach that accelerates inner loops via runtime binary profiling is presented in Reference [34]. Profiling is performed by auxiliary hardware monitoring the execution stream for backward branches. The detected hot BBs are modulo-scheduled via software, using binary translation tools that reside in the code memory of the host processor onto the proposed mesh array. Either RISC or VLIW binary can be targeted, with support for RAW/WAR dependency detection. The BBs contain between 40 and 120 instructions. The array contains 16 heterogeneous FUs, supporting logic and integer arithmetic operations, as well as up to two memory accesses, conditional assignments and evaluation of loop exit points. Floating-point operations are not supported. Each FU contains a local register file, and communicates with other units via a shared crossbar. The accelerator is loosely coupled to the main processor and fetches operands from its register file, and both shared the same data memory. The modulo scheduling algorithm is up to three orders of magnitude faster when compared to compilation targeting VLIW with several issue-widths, for the 11 loops detected by profiling four benchmarks. The average ILP achieved is close to the maximum possible ILP when scheduling for an array capable of two memory accesses per clock cycle. When implemented on a Virtex-6 FPGA the accelerator requires 13,000 LUTs and 23 *Block RAM*s (BRAMs), which is 1.3× smaller than a 8-issue VLIW, over which it achieves an average speedup of 2× for the 11 detected loops containing between 50 to 120 instructions.

Binary Translation – Segment Type *A1*, Detection *B3*, Translation *C3*, Binary Mod. *D3*, Acc. Type *E1*

Acc. Architecture – Interface *A4*, Arrangement *B3*, Interconnects *C4*, Ops. *D2*, Mem. Access *E4*, Exec. Model *F4*

4.8.6 ASTRO. Lin et al. [56] detail an approach focused on achieving acceleration by maximizing memory access parallelism. MicroBlaze instruction sequences are detected by simulated execution profiling, followed by synthesis of one loop accelerator per instruction sequence. The accelerators are direct translations of CDFGs into pipelined datapaths coupled to the MicroBlaze via a peripheral bus. Migration of execution is performed by monitoring the MicroBlaze's instruction address at runtime. Dynamic memory access analysis is performed to determine disjoint regions of access. Memory accesses within the hot regions are grouped into partitions based on access dependencies. Each partition is assigned a customized cache. The analysis also deals with data hazards. With this information, a tailored BRAM-based multi-cache system is generated per-accelerator, allowing for efficient exploration of data parallelism. The cache system for the accelerators may require, at least, a partial cache invalidation after accelerator execution, leading to overheads. A typical accelerator with a multi-cache network requires 20,400 LUTs and 5,900 FFs on a Virtex-5 device. For 10 benchmarks from MiBench and SPEC2006, the geometric mean speedup achieved was 7.1× versus software-only execution. Finally, maximizing concurrent accesses leads to an average speedup of 2× over single-access accelerators, approximately.

Binary Translation – Segment Type *A2*, Detection *B2*, Translation *C2*, Binary Mod. *D3*, Acc. Type *E3*

Acc. Architecture – Interface *A4*, Arrangement *B4*, Interconnects *C1*, Ops. *D2*, Mem. Access *E4*, Exec. Model *F3*

4.8.7 Malazgirt et al. In Reference [59] a VLIW processor is customized offline, based on profiled execution traces. Each target instruction trace is split into two, one contains only memory access instructions, and a second contains the remaining instructions. The former is analysed so that a schedule is generated that maximizes ILP by packing as many memory accesses into the same VLIW instruction as possible. The aim is to reduce resource under-utilization in VLIWs, due to inefficient exploitation of ILP. A second optimization pass effectively minimizes the difference between the maximum and average ILP, which minimizes the number of required FUs (i.e., issue-width) and memory banks. The memory bank architecture is such that each bank contains multiple memories with the same content. Each bank is associated to each FU, providing multiple read ports and a single write port. An evaluation of 11 string matching algorithms on a simulator yields an average speedup of 3× when comparing the customized VLIWs with a baseline RISC processor modeled off the parameters retrieved for a MicroBlaze synthesized for a Kintex-7 FPGA operating at 100 MHz. The resulting power consumption is 2.8× higher for the VLIW design.

Binary Translation – Segment Type *A1*, Detection *B2*, Translation *C2*, Binary Mod. -, Acc. Type *E2*

Acc. Architecture – Interface -, Arrangement *B3*, Interconnects *C3*, Ops. *D3*, Mem. Access *E3*, Exec. Model *F2*

4.8.8 Rokicki et al. A system that fully translates the MIPS binary of a target application to a 4-issue VLIW is presented in Reference [77]. During the initial stages of execution, a first-pass binary translation implemented in hardware retargets the code, and also instruments it for segment detection in a second phase. Instrumentation is possible by taking advantage of free slots in the VLIW instructions, thus minimizing execution interference. Supported hot regions are composed of sequences of BBs with up to 256 instructions, which are generated from profile data by another hardware module. A final step generates an optimized sequence of VLIW instructions that exploit loop pipelining. The hardware modules composing the dynamic binary translation engine were developed in *C* and then generated via HLS. Validation targeted 65 nm technology operating at 250 MHz. For 10 kernels from the Mediabench suite [54], an average speedup of 3× is possible by pipelining hot code regions, over the non-accelerated VLIW code generated by the first pass of the binary translation. Relying on hardware-based binary translation can be nearly one order of magnitude faster than implementing the same functionality through software. The pipelining step is approximately 5.5× faster when accelerated through hardware and 11.0× more energy efficient.

Binary Translation – Segment Type *A2*, Detection *B3*, Translation *C3*, Binary Mod. *D3*, Acc. Type *E2*

Acc. Architecture – Interface -, Arrangement *B3*, Interconnects *C3*, Ops. *D4*, Mem. Access *E3*, Exec. Model *F2*

5 SUMMARY, COMPARISON, AND CLASSIFICATION

Table 3 provides an overview of a selection of the representative approaches presented in this article. It compiles information on the most important aspects of these approaches, namely the architecture of the accelerators, the operations they support, memory access capabilities, which type of binary code segments are targeted, and the translation/mapping methods. Finally, we attempted to retrieve speedup and energy consumption data for all cases. Speedups are shown as reported, meaning that some speedups are arithmetic means, and others are geometric means (shown in footnotes). We also note that these speedups are not necessarily comparable, given the different baselines, which are shown in the second column, *CPU*. In some cases, the *CPU* is

Table 3. Overview of Binary Acceleration Approaches

Work	CPU	Segment Type	Detection and Translation	Acc. Architecture & Coupling	Supported Operations	Memory Access	Speedup	*E* Reduc.
Warp [58, 84, 92]	MicroBlaze	Hot BBs	Online detection, disassembly and circuit synthesis	Custom fine-grained bit-level logic; loose	Fixed-point arithmetic; logic	One regular pattern access per cycle	3.2×[1]	2.9×
ADEXOR [67, 68]	MIPS	Multi-exit BB traces	Offline profiling, binary modification and config. generation	Heterogeneous rows of different widths; feed-forward crossbars; tight	Fixed-point arithmetic (no div. or mul.); logic	One store per instruction via the host	1.9×[2]	1.3×
DIM [8, 80]	MIPS	Sequences of BBs	Runtime binary profiling and acc. config. generation	Heterogeneous FU rows; rich forward connectivity; tight	Int. arithmetic (exclud. div.); logic	Unspecificed # of arb.[3] accesses per row	2.6×[1]	2.2×
CCA [17–19]	ARM	BB or SuperBlock sub-graphs	Compile time detection; Runtime config. generation	2 alternating FU types in rows; inter-row crossbars; tight	Int. arithmetic (no div. or mul.); logic	Not supported	2.2×[2]	N/A
DySE [40]	SPARC[4]	Frequent single-exit BB traces	Compile-time profiling and binary modification	Heterogeneous FU mesh w/ neighbour connections; tight	Logic; int. and floating-point arithmetic	Indirectly via host CPU instructions	2.2×[1]	1.7×[1]
BERET [44]	ARM	Looping Superblocks	Compile-time detection and acc. generation	Several sub-graph accelerators sharing a register file; tight	Integer arithmetic; logic	Write access via shared data cache	1.1×[2]	1.5×
CLA [72–74]	MicroBlaze	Megablocks	Offline detection (via simulation) and acc. generation	Single heterogeneous modulo-scheduled row w/ tailored interconnect; loose	Logic; int. and floating-point arithmetic	2 arb.[3] accesses per cycle to shared memory	5.6×[1]	3.9×
PLA [31, 32]	OR-1200[10]	Loop bodies from static binary	Compile-time customization of acc. template	Single heterogeneous modulo-scheduled row w/ tailored interconnect; loose[4]	Integer arithmetic; logic	1 arb.[3] access to a local memory	12.0×[5,2]	12.0×[5]
PPA [71]	ARM[10]	Loops or sequences of static binary	Manual detection and compile-time modulo-scheduling	Multiple homogeneous cores, with 4 ALUs and register file	Integer arithmetic	1 access per column to shared global memory	2.5×[6]	8.3×[7]
Paek et al. [70]	ARM	Static loops w/known trip count and stride	Offline binary dissassembly and config. generation	Mesh of homogeneous ALUs; neighbour or global connections; loose	Integer arithmetic[4]	1 wide sequential access to shared scratchpad	9.4×[2,8] / 1.5×[9]	N/A
Chen et al. [13]	ARM	Frequent static instruction sequences	Compile time detection and translation into custom instructions	Three heterogeneous FUs; feed-forward connectivity; tight	Integer arithmetic[4]; logic	Not supported	1.1×[2]	N/A
Ferreira et al. [34]	RISC VLIW	Hot BBs	Runtime trace detection and modulo-scheduling	16 FUs w/local register files; global crossbar; loose	Int. arithmetic; logic; conditional assignments	2 accesses per clock cycle to shared memory	2.0×[2]	N/A

(Continued)

Table 3. Continued

Work	CPU	Segment Type	Detection and Translation	Acc. Architecture & Coupling	Supported Operations	Memory Access	Speedup	*E* Reduc.
ASTRO [56]	MicroBlaze	Frequent trace sequences of BBs	Offline detection (by simulation) and synthesis of acc. and tailored cache	Rows of single-function units; tailored inter-row connections; loose	Integer arithmetic; logic	Multiple arb.[3] accesses to custom shared cache	7.1×[1]	1.6×
Malazgirt et al. [59]	MicroBlaze[10]	Instruction traces (unspecified type)	Compile time customization of VLIW based on ILP maximization	Tailored VLIW instance; crossbar connectivity	Logic; branches; int. and floating-point arithmetic	Multiple arb.[3] accesses to custom multi-port memory	3.0 ×[2]	1.1×
Rokicki et al. [77]	MIPS[10]	BB traces w/ up to 256 instructions	Runtime profiling, modulo-scheduling, and binary re-writing	Custom VLIW architecture	Vex-based instruction set	Yes	3.3×[2]	11.0×[11]

[1]geometric mean, [2]arithmetic mean, [3]arbitrary, [4]presumed, [5]inferred from other data, [6]over PPA execution w/o modulo scheduling, [7]over ARM baseline for a single application, [8]kernels only, [9]or one application, [10]used only as baseline, [11]over software-based binary translation.

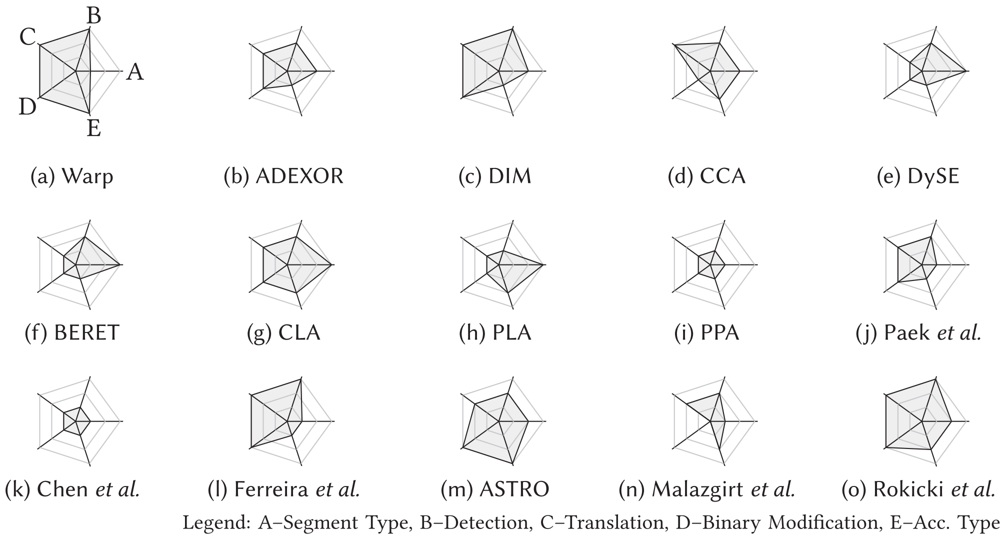
the host processor to the accelerator, and in others, the proposed system does not rely on a main processor, in which case the architecture shown in *CPU* represents the baseline. The last column shows the energy consumption reduction.

As shown in Table 3, the preferred processors to serve as host/baseline are RISC architectures, mainly ARM, MIPS, or the MicroBlaze. Since the approaches presented here are based on binary translation, which entails interpretation of the binary code (potentially at runtime), adopting reduced instruction sets simplifies the implementation. If these approaches are ever to target other architectures, e.g., by relying on recent hybrid devices such as Intel's processor family with integrated FPGA [48], then some effort may be required to support CISC architectures. Two thirds of the cases perform the binary translation steps prior to runtime, either as a compile-time or post-compile step, along the generation and/or configuration of the accelerator hardware. The remaning cases perform all necessary steps at runtime, and only one case, the CCA, supports one operating mode where binary profiling is performed offline, followed by runtime execution migration.

As for the accelerator architecture, six approaches opt for a row-based architecture, four for a mesh design, and the remaining six fall into mixed categories (e.g., Warp's fine-grained fabric, or BERET's sub-graph engines). In terms of operations, nine architectures support only integer arithmetic, and five support either fixed- or floating-point arithmetic. The choice of interface between accelerator and host varies, with six implementations opting for a tight coupling, and six for a loose coupling; the remaining cases are standalone processing engines without a host processor. With the exception of one case, memory access is always supported in some fashion, with two cases supporting memory access only in a partial fashion.

Some data on speedups and energy consumption is unavailable. Some speedups are reported via an arithmetic mean and others via a geometric mean. We derived geometric mean values where individual values for segment/application speedups were given. Despite this, it is expected and possible to observe that, generally, the attained speedups are smallest for approaches based on an ARM processor, followed by MIPS, and greatest for those based on the MicroBlaze. Conversely, the energy consumption reduction is greater when employing an ARM and smallest for MicroBlaze.

Figure 16 shows another overview of all approaches through radar charts, according to the presented taxonomy axes (see Table 1). Roughly half of the approaches do not modify the host



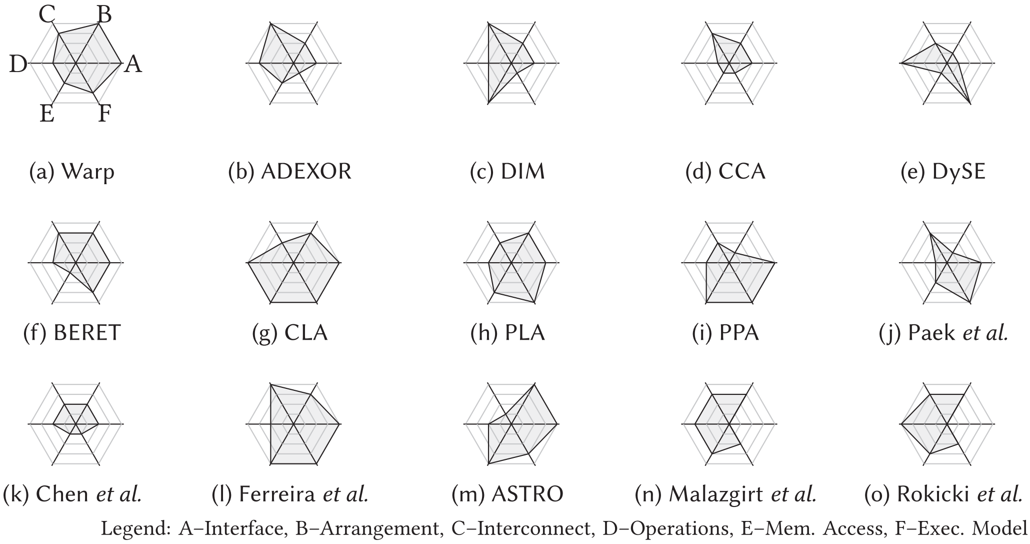
(a) Warp (b) ADEXOR (c) DIM (d) CCA (e) DySE

(f) BERET (g) CLA (h) PLA (i) PPA (j) Paek *et al.*

(k) Chen *et al.* (l) Ferreira *et al.* (m) ASTRO (n) Malazgirt *et al.* (o) Rokicki *et al.*

Legend: A–Segment Type, B–Detection, C–Translation, D–Binary Modification, E–Acc. Type

Fig. 16. Overview of the binary translation characteristics of the reviewed approaches (full Legend in Table 1).



(a) Warp (b) ADEXOR (c) DIM (d) CCA (e) DySE

(f) BERET (g) CLA (h) PLA (i) PPA (j) Paek *et al.*

(k) Chen *et al.* (l) Ferreira *et al.* (m) ASTRO (n) Malazgirt *et al.* (o) Rokicki *et al.*

Legend: A–Interface, B–Arrangement, C–Interconnect, D–Operations, E–Mem. Access, F–Exec. Model

Fig. 17. Overview of the accelerator architecture features of the reviewed approaches (full legend in Table 2).

application binary when compared to deploying the same binary to a non-accelerated system based on the same host processor. However, all systems perform some type of binary translation to target the accelerator, mostly during an offline stage. The accelerator instance is specialized for the target binary segments to accelerate by half of the approaches, with only the Warp approach doing so at runtime. Nearly all approaches rely on traces over static binary analysis, and most perform the analysis prior to runtime. Finally, the most targeted type of binary segment are *Basic Block*s, representing inner loops, with few approaches using sequences of blocks or loop traces.

Figure 17 provides an equivalent view of the accelerator architecture (see Table 2). Most approaches support only full integer arithmetic, supplemented by logic operations. The majority also implement at least limited memory access, with one third supporting multiple concurrent accesses to arbitray addresses. As for the execution model, the approaches either rely on the least complex type of acceleration, which is to target sequential instruction sequences expressed as custom instructions or on the most complex approach of targeting large traces, which require more complex architectures and temporal/spatial scheduling of operations to achieve loop pipelining. The interconnectivity is often reduced, maintaining some reconfigurability tailored according to either an analysis of representative segments or according to the specific target segments. Finally, half of the approaches opt for a tightly coupled accelerator and the remaining for a loose connection.

In summary, there does not seem to be a preferred approach in terms of either type of segment or accelerator architecture. We can, however, make the following observations. First, we compare execution strategies for acyclic and cyclic segments.

Acyclic segments, usually smaller, are more efficiently accelerated in approaches with a tight coupling. This avoids data transfer overheads, especially since the volume of data that this type of segment consumes and produces is small. The advantage of this general approach, is that the acceleration gains can be spread out throughout the program. This means that applications without very obvious hotspots, like large loops with memory accesses, can be accelerated to some degree. Even pre-compiled library code, linked into the user application, may be accelerated. Finally, the technology for JIT re-compilation of binary is very mature, with the exception that it is not usually used to target accelerators. This gives this acceleration strategy more potential for ubiquity. The most apt accelerator architecture for this are small row-based arrays with feed-forward connectivity, where CDFG nodes and edges can be easily mapped, facilitating runtime re-targeting.

In contrast, acceleration of cyclic segments requires implementation of some control operations regardless of loop size. However, gains from acceleration of small loops like hot BBs are limited, since there is little ILP to exploit in such small instruction windows. Instead, small loops can be indirectly accelerated by the aforementioned tightly integrated designs, where the bulk of the loop is translated, and the remaining control (or unsupported) instructions are left on the processor side. This means that loosely coupled meshes and pipeline type accelerators are best suited for larger loops, since more ILP can be exploited due to the larger segment size and by loop pipelining. Since larger loops almost always operate on multiple sources of data, accelerator designs capable of multiple memory accesses are essential to achieve significant speedups. This is only possible with a loose coupling that does not restrict the accelerator to operating on the host processor register file. Finally, row-based loop accelerators are essentially pipelined datapaths and therefore scale poorly for large CDFGs. Altough they require less complex translation, that requires no temporal partitioning, mesh arrays supported by modulo-scheduling can in theory support loops of any size. The only pratical limits are the scheduling time and storage for configuration data.

Second, a small number of approaches [59, 77] rely on specializing a generic processor architecture rather than interfacing an existing host processor with additional hardware. Approaches where a tightly coupled accelerator is controlled by custom instructions are similar to this method. In this regard, they can be considered automated Instruction Set Extension approaches [36].

Finally, we find that in terms of use, these types of computing architectures, at least in their present state of development, are fit for specific bare-metal embedded scenarios. To the best of our knowledge, there has been no significant work on methods for supporting either generation or migration of workload to such accelerators in embedded systems running an operating system. The main goal is to achieve acceleration of a single-application by exploitation of data-independent operations at specific points rather than accelerating an entire application by gearing the hardware design toward higher operating frequencies. In this context, a single application, or at most a particular application domain, benefits the most from the specialization provided. It should also be noted that accelerating binary segments resorting to the presented techniques does not expose large amounts of data parallelism that lead to implementations that exploit vectorization. One exception to point out is ASTRO [56], which specifically maximizes data bandwidth, but even this case is focused on intra-iteration data parallelism through many-port memories rather than full parallelization of data acesses that are independent across iterations.

6 CONCLUSION

This survey reviewed representative approaches regarding automatic acceleration of applications via migrating the respective binary code to specialized accelerator devices. Most work has been published within the past 10 years. The approaches are based on offloading the effort of optimizing the target application, ideally in terms of performance and energy consumption. This is done by automating the generation of custom accelerator hardware, automatically offloading computation to the accelerator hardware, or both. Compilation flows or runtime techniques capable of targeting heterogeneous computing elements, preferably with minimal developer intervention, are proposed to accomplish this. To provide functional validations and experimental evaluation, ASIC or FPGAs implementations, along with simulations, are used. We find that a platform for this type of approach, i.e., for automated hardware generation or runtime reconfiguration, is yet to be realized, especially if the process is to be offloaded to a runtime environment.

There is, however, a trend toward hardware/software co-design and reconfigurable devices. For instance, FPGAs have evolved into fully fledged *System-on-a-chip*s (SoCs) that contain hardcore processors and specialized hardware modules (e.g., memory controllers, encryption modules, *etc.*), with a software interface to the reconfigurable logic [102]. This thus addresses one of the highlighted issues: the integration of custom hardware with the host processor. Simultaneously, Intel's new processors containing an integrated FPGA [49] are a step toward the same direction, potentially standardizing how reconfigurable logic and host processor integrate. In addition, this integration may allow for the detection and translation techniques proposed by the presented approaches to be offloaded fully to runtime. Given this, we believe that the reviewed binary acceleration techniques fulfill the required infrastructure of such future fully transparent acceleration approaches and that they demonstrate the potential for improvement of performance and energy consumption through hardware specialization.

REFERENCES

[1] E. R. Altman, D. Kaeli, and Y. Sheffer. 2000. Welcome to the opportunities of binary translation. *Computer* 33, 3 (Mar. 2000), 40–45.

[2] José Carlos Alves and Pedro C. Diniz. 2011. Custom FPGA-based micro-architecture for streaming computing. In *Proceedings of the Southern Conference on Programmable Logic*. 51–56.

[3] João Andrade, Nithin George, Kimon Karras, David Novo, Vitor Silva, Paolo Ienne, and Gabriel Falcão. 2015. From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–8.

[4] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. 94–103.

[5] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. 2003. Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations. In *Proceedings of the Workshop on Application Specific Processors, Held in Conjunction with the International Symposium on Microarchitecture*.

[6] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Yun Wang, and Y. Zemach. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium/spl reg/-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture 2003 (MICRO-36)*. 191–201.

[7] L. Bauer, M. Shafique, and J. Henkel. 2011. Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors. In *Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS'11)*. 80–87.

[8] Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. 2008. Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition.* 1208–1213.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* 72–81.

[10] João Bispo and João M. P. Cardoso. 2010. On identifying and optimizing instruction sequences for dynamic compilation. In *Proceedings of the International Conference on Field-Programmable Technology.* 437–440.

[11] Lilian Bossuet, Michael Grand, Lubos Gaspar, Viktor Fischer, and Guy Gogniat. 2013. Architectures of flexible symmetric key crypto engines-a survey: From hardware coprocessor to multi-crypto-processor system on chip. *Comput. Surv.* 45, 8 (2013).

[12] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.* 42, 4, Article 13 (June 2010), 65 pages.

[13] Liang Chen, Joseph Tarango, Tulika Mitra, and Philip Brisk. 2013. A just-in-time customizable processor. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design.* 524–531.

[14] Shaoyi Cheng, Mingjie Lin, Hao Jun Liu, Simon Scott, and John Wawrzynek. 2012. Exploiting memory-level parallelism in reconfigurable accelerators. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines.* 157–160.

[15] Kiyoung Choi. 2011. Coarse-grained reconfigurable array: Architecture and application mapping. *IPSJ Trans. Syst. LSI Des. Methodol.* 4 (2011), 31–46. https://doi.org/10.2197/ipsjtsldm.4.31.

[16] C. Cifuentes and M. Van Emmerik. 2000. UQBT: Adaptable binary translation at low cost. *Computer* 33, 3 (Mar. 2000), 60–66.

[17] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztián Flautner. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the International Symposium on Computer Architecture.* 272–283.

[18] Nathan Clark, Amir Hormati, and Scott Mahlke. 2008. VEAL: Virtualized execution accelerator for loops. In *Proceedings of the International Symposium on Computer Architecture.* 389–400.

[19] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztián Flautner. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the International Symposium on Microarchitecture.* 30–40.

[20] Katherine Compton and Scott Hauck. 2002. Reconfigurable computing: A survey of systems and software. *Comput. Surv.* 34, 2 (June 2002), 171–210.

[21] Standard Performance Evaluation Corporation. 2006. *SPEC CPU Benchmark Suites.* www.spec.org/cpu.

[22] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. 1999. Architecture design of reconfigurable pipelined datapaths. In *Proceedings of the Anniversary Conference on Advanced Research in VLSI.* 23–40.

[23] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. CPU DB: Recording microprocessor history. *Queue* 10, 4, Article 10 (Apr. 2012), 18 pages.

[24] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2016. Optimizing indirect branches in dynamic binary translators. *ACM Trans. Archit. Code Optim.* 13, 1, Article 7 (Apr. 2016), 25 pages.

[25] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. 2018. Transformations of high-level synthesis codes for high-performance computing. *CoRR* abs/1805.08288 (2018).

[26] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circ.* 9, 5 (Oct. 1974), 256–268.

[27] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Trans. Comput.* 50, 6 (Jun. 2001), 529–548.

[28] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro* 32, 3 (May 2012), 122–134.

[29] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.* 30, 3 (Aug. 2012), 11:1–11:27.

[30] Jair Fajardo, Mateus B. Rutzig, Luigi Carro, and Antonio Carlos S. Beck. 2013. Towards a multiple-ISA embedded system. *J. Syst. Arch.* 59, 2 (Feb. 2013), 103–119.

[31] Kevin Fan, Manjunath Kudlur, Ganesh Dasika, and Scott Mahlke. 2009. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture.* 313–322.

[32] Kevin Fan, Hyunchul Park, Manjunath Kudlur, and Scott Mahlke. 2008. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization.* 124–133.

[33] E. Fatehi and P. V. Gratz. 2014. ILP and TLP in shared memory applications: A limit study. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*. 113–125.

[34] Ricardo Ferreira, Waldir Denver, Monica Pereira, Jorge Quadros, Luigi Carro, and Stephan Wong. 2014. A run-time modulo scheduling by using a binary translation mechanism. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 75–82.

[35] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. 1992. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA.

[36] Carlo Galuzzi and Koen Bertels. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfig. Technol. Syst.* 4, 2 (May 2011), 18:1–18:28.

[37] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Reed. 1999. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the International Symposium on Computer Architecture*. 28–39.

[38] Ann Gordon-Ross and Frank Vahid. 2005. Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Trans. Comput.* 54, 10 (Oct. 2005), 1203–1215.

[39] Chen-Han Govindaraju, Venkatramanand Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy efficient computing. *IEEE Micro* 32, 5 (Sep. 2012), 38–51.

[40] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 503–514.

[41] The IMPACT Research Group. 2012. Parboil benchmark suite. Retrieved January 22, 2019 from http://impact.crhc.illinois.edu/parboil/parboil.aspx.

[42] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. 2000. Dynamic and transparent binary translation. *Computer* 33, 3 (Mar. 2000), 54–59.

[43] P.K. Gupta. 2016. Accelerating Datacenter Workloads. Keynote at FPL2016. Retrieved July 26, 2019 from https://www.fpl2016.org/.

[44] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 12–23.

[45] Reiner Hartenstein. 2001. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*.564–570.

[46] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1 (May 1993), 229–248.

[47] Texas Instruments. 2002. TMS320C6000 Image Library. Retrieved January 22, 2019 from www.ti.com/tool/sprc264.

[48] Intel. 2006. Intel Processors and FPGAs—Better Together. Retrieved January 22, 2019 from https://itpeernetwork.intel.com/intel-processors-fpga-better-together.

[49] Intel. 2019. Intel FPGA SDK for OpenCL Software Technology. Retrieved July 26, 2019 from https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[50] Yangsu Kim, Kyuseung Han, and Kiyoung Choi. 2011. A host-accelerator communication architecture design for efficient binary acceleration. In *Proceedings of the International SoC Design Conference*. 361–364.

[51] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2011. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Trans. Des. Autom. Electr. Syst.* 16, 4 (Oct. 2011), 42:1–42:27.

[52] Monica S. Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 318–328.

[53] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 175–183.

[54] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 330–335.

[55] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. 2012. High-level synthesis: Productivity, performance, and software constraints. *J. Electr. Comput. Eng.* 2012, 2 (2012). https://www.hindawi.com/journals/jece/2012/649057/cta/.

[56] Mingjie Lin, Shaoyi Chen, Ronald DeMara, and John Wawrzynek. 2015. ASTRO: Synthesizing application-specific reconfigurable hardware traces to exploit memory-level parallelism. *Microprocess. Microsyst.* 39, 7 (2015), 553–564.

[57] Roman Lysecky and Frank Vahid. 2004. A configurable logic architecture for dynamic hardware/software partitioning. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1. 480–485.

[58] Roman Lysecky and Frank Vahid. 2009. Design and implementation of a MicroBlaze-based warp processor. *ACM Trans. Embed. Comput. Syst.* 8, 3 (Apr. 2009), 22:1–22:22.

[59] Gorker Alp Malazgirt, Arda Yurdakulm, and Small Niar. 2015. Customizing VLIW processors from dynamically profiled execution traces. *Microprocess. Microsyst.* 39, 8 (2015), 656–673.

[60] Afzal Malik, B. Moyer, and D. Cermak. 2000. A lower power unified cache architecture providing power and performance flexibility. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design.* 241–243.

[61] S. A. Manavski. 2007. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Proceeedings of the 2007 IEEE International Conference on Signal Processing and Communications.* 65–68.

[62] Arash Mehdizadeh, Behnam Ghavami, Mortza Saheb Zamani, Hossein Pedram, and Farhad Mehdipour. 2007. An efficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor. In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration.* 151–156.

[63] Gayatri Mehta, Justin Slander, Mustafa Baz, Brady Hunsaker, and Alex K. Jones. 2007. Interconnect customization for a coarse-grained reconfigurable fabric. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*1–8.

[64] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (Jun. 2007), 89–100.

[65] Nuno Neves, Pedro Tomás, and Nuno Roma. 2015. Efficient data-stream management for shared-memory many-core systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications.* 1–8.

[66] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (Mar. 2008), 40–53.

[67] Hamid Noori, Farhad Mehdipour, Koji Inoue, and Kazuaki Murakami. 2012. Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization. *J. Supercomput.* 60, 2 (May 2012), 196–222.

[68] Hamid Noori, Farhad Mehdipour, Kazuaki Murakami, Koji Inoue, and Morteza Saheb Zamani. 2008. An architecture framework for an adaptive extensible processor. *J. Supercomput.* 45, 3 (Sep. 2008), 313–340.

[69] University of Michigan. 2013. MiBench benchmark suite. Retrieved January 22, 2019 from http://vhosts.eecs.umich.edu/mibench.

[70] Jong Kyung Paek, Kiyoung Choi, and Jongeun Lee. 2011. Binary acceleration using coarse-grained reconfigurable architecture. *SIGARCH Comput. Arch. News* 38, 4 (Jan. 2011), 33–39.

[71] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.* 370–380.

[72] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2014. A reconfigurable architecture for binary acceleration of loops with memory accesses. *ACM Trans. Reconfig. Technol. Syst.* 7, 4 (Dec. 2014), 29:1–29:20.

[73] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2017. Generation of customized accelerators for loop pipelining of binary instruction traces. *IEEE Trans. VLSI Syst.* 25, 1 (Jan. 2017), 21–34.

[74] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2019. Dynamic partial reconfiguration of customized single-row accelerators. *IEEE Trans. VLSI Syst.* 27, 1 (Jan. 2019), 116–125.

[75] Tim Peters. 1992. Livermore loops coded in C. Retrieved January 22, 2019 from www.netlib.org/benchmark/livermorec.

[76] B. Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the Annual International Symposium on Microarchitecture.* 63–74.

[77] Simon Rokicki, Erven Rohou, and Steven Derrien. 2017. Hardware-accelerated dynamic binary translation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition.* 1062–1067.

[78] S. Rokicki, E. Rohou, and S. Derrien. 2019. Hybrid-DBT: Hardware/software dynamic binary translation targeting VLIW. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 38, 10 (2019), 1872–1885. https://ieeexplore.ieee.org/document/8429242.

[79] Mateus Beck Rutzig, Antonio Carlos S. Beck, and Luigi Carro. 2013. A transparent and energy aware reconfigurable multiprocessor platform for simultaneous ILP and TLP exploitation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition.* 1559–1564.

[80] Mateus Beck Rutzig, Antonio Carlos S. Beck, Felipe Madruga, Marco A. Alves, Henrique C. Freitas, Nicolas Maillard, Philippe O. A. Navaux, and Luigi Carro. 2011. Boosting parallel applications performance on applying DIM technique in a multiprocessing environment. *Int. J. Reconfig. Comput.* 2011 (Jan. 2011), 4:1–4:13. https://www.hindawi.com/journals/ijrc/2011/546962/cta.

[81] Junaid Shuja, Abdullah Gani, Kashif Bilal, Samee Khan, Sajjad Madani, Atta Ur Rehman Khan, and Albert Zomaya. 2016. A survey of mobile device virtualization: Taxonomy and state-of-the-art. *Comput. Surv.* 49, 4 (2016), 1.

[82] Mihai Sima, Michael McGuire, and Julien Lamoureux. 2009. Coarse-grain reconfigurable architectures—Taxonomy. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. 975–978.

[83] Hertej Singh, Ming-Hau Lee, Guangming Lu, Fadi. J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* 49, 5 (May 2000), 465–481.

[84] Greg Stitt and Frank Vahid. 2011. Thread warping: Dynamic and transparent synthesis of thread accelerators. *ACM Trans. Des. Autom. Electr. Syst.* 16, 3 (Jun. 2011).

[85] Mirjana Stojilović, David Novo, Lazar Saranovac, Philip Brisk, and Paolo Ienne. 2013. Selective flexibility: Creating domain-specific reconfigurable arrays. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 32, 5 (May 2013), 681–694.

[86] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* 12, 3 (2010), 66–73.

[87] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[88] M. B. Taylor. 2013. A landscape of the new dark silicon design regime. *IEEE Micro* 33, 5 (Sep. 2013), 8–19.

[89] Jürgen Teich. 2012. Hardware/software codesign: The past, the present, and predicting the future. In *Proceedings of the IEEE*, Vol. 100. 1411–1430. https://ieeexplore.ieee.org/document/6172642.

[90] EEMBC The Embedded Microprocessor Benchmark Consortium. 2015. CoreMark-Pro. Retrieved January 22, 2019 from http://www.eembc.org.

[91] Trimaran. 2007. An infrastructure for research in backend compilation and architecture exploration. Retrieved January 22, 2019 from www.trimaran.org.

[92] Frank Vahid, Greg Stitt, and Roman Lysecky. 2008. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer* 41, 7 (2008), 40–46.

[93] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. Static versus scheduled interconnect in coarse-grained reconfigurable arrays. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 268–275.

[94] Stamatis Vassiliadis, Stephan Wong, and Sorin Cotöfană. 2001. The MOLEN $\rho\mu$-coded processor. In *Field-Programmable Logic and Applications*, Gordon Brebner and Roger Woods (Eds.). Springer, Berlin, 275–285.

[95] David W. Wall. 1999. Limits of instruction-level parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'99)*.

[96] J. Wang, J. Pang, X. Liu, F. Yue, J. Tan, and L. Fu. 2019. Dynamic translation optimization method based on static pre-translation. *IEEE Access* 7 (2019), 21491–21501. https://ieeexplore.ieee.org/document/8635523.

[97] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From hack to elaborate technique – A survey on binary rewriting. *ACM Comput. Surv.* 52, 3, Article 49 (Jun. 2019), 37 pages.

[98] Steven Wilton, Noha Kafafi, Mei Bingfeng, and Serge Vernalde. 2004. Interconnect architectures for modulo-scheduled coarse-grained reconfigurable arrays. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. 33–40.

[99] Wayne Wolf. 2003. A decade of hardware/software codesign. *Computer* 36, 4 (2003), 38–43.

[100] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Arch.* 24–36.

[101] Xilinx. 2019. SDAccel Development Environment. Retrieved July 26, 2019 from https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[102] Xilinx. 2019. Zynq UltraScale+ MPSoC. Retrieved July 26, 2019 from https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[103] Pan Yu and Tulika Mitra. 2004. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of the Annual Design Automation Conference*. 723–728.

[104] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. 129–140.

[105] Vojin Zivojnovic, Juan M. Velarde, Christian Schlager, and Heinrich Meyr. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*.

Received February 2019; revised September 2019; accepted October 2019