Software Transparent Dynamic Binary Translation for Coarse-Grain Reconfigurable Architectures

Matthew A. Watkins
Lafayette College
Easton, PA

Tony Nowatzki
Univ. of Wisconsin-Madison
Madison, WI

Anthony Carno
Virginia Tech
Blacksburg, VA

ABSTRACT

The end of Dennard Scaling has forced architects to focus on designing for execution efficiency. Course-grained reconfigurable architectures (CGRAs) are a class of architectures that provide a configurable grouping of functional units that aim to bridge the gap between the power and performance of custom hardware and the flexibility of software. Despite their potential benefit, CGRAs face a major adoption challenge as they do not execute a standard instruction stream.

Dynamic translation for CGRAs has the potential to solve this problem, but faces non-trivial challenges. Existing attempts either do not achieve the full power and performance potential CGRAs offer or suffer from excessive translation time. In this work we propose DORA, a Dynamic Optimizer for Reconfigurable Architectures, which achieves substantial (2X) power and performance improvements while having low hardware and insertion overhead and benefiting the current execution. In addition to traditional optimizations, DORA leverages dynamic register information to perform optimizations not available to compilers and achieves performance similar to or better than CGRA-targeted compiled code.

1. INTRODUCTION

The end of Dennard Scaling has led to serious power constraints in today's microprocessors. The main focus for architects is now execution efficiency, achieving higher performance for the same power or the same performance with less power. Coarse-grained reconfigurable architectures (CGRAs) provide parallelism through a configurable grouping of functional units [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and strive to approach the power and performance of custom hardware while maintaining the flexibility of software. This flexibility makes them potentially more attractive than existing options like SIMD units whose rigidity limits their scope and benefit.

A major challenge of CGRAs is that they require modification of the software. Most previous CGRA work relies either on the programmer or compiler to perform this modification [3, 11, 12, 13]. Both options have multiple disadvantages, including i) requiring access to the source code and recompilation (which can still miss regions like library code), ii) requiring recompilation/redesign anytime the array architecture is changed, as is likely to happen with future generations, iii) requiring multiple versions of the code to support multiple architectures, and iv) potentially long adoption times from when capabilities are first released until when mainstream programs adopt them. CGRAs further lack a formalized abstraction to expose their architecture to the ISA.

Dynamic binary translation (DBT) translates and optimizes a program while it is executing. DBT has the potential to overcome all of the aforementioned mentioned issues, allowing CGRAs to be entirely transparent to software and provide immediate benefits to all programs. Further, DBT can use runtime information to perform optimizations not possible at compile time and revert to a previous version when an optimization yields no benefit. In certain environments, such as mobile SoCs, an exposed hardware/software solution, which introduces practical challenges such as software design and long-term support, can be a very difficult sell, essentially necessitating a software-transparent solution for business reasons.

An ideal CGRA DBT system would have three key elements. First, the targeted CGRA could provide power and performance benefits significant enough to warrant the added complexities. Second, system performance would match, or exceed, the performance achieved by offline compilation so as not to leave potential benefits unrealized. Finally, the system would perform optimization and mapping quickly enough to have minimal power overhead and achieve near optimal performance benefit for the current execution of even relatively short computation. To our knowledge, no existing work provides all three of these features. The majority of existing CGRA DBT proposals target 1-D feed-forward arrays and provide relatively modest average performance benefits of 9-42% [14, 15, 16, 17] compared to the 2-10X benefits reported for some CGRAs [18, 19, 20]. Further, as discussed in Section 6.4, compiler generated SIMD code already achieves 30% improvements, greatly diminishing the appeal of these existing DBT works as current technology already provides similar benefits and forgoes the added complexity. Other works that dynamically target reconfigurable architectures, which are discussed in detail in Section 7, similarly face shortcomings in one or more of the aforementioned areas.

This work proposes DORA (a Dynamic Optimizer for (Coarse-Grained) Reconfigurable Architectures), the first

978-1-4673-9211-2/16/$31.00

system that simultaneously achieves all three of the above elements. DORA improves performance and reduces energy consumption by 2X, matches or exceeds the performance of a sophisticated offline compiler, and performs optimization and mapping quickly enough to achieve near optimal benefit for even relatively short runs. We identify two key insights that make this possible. First, achieving significant performance and power benefits with manageable translation overheads requires the right balance in array architecture. We find that sparsely connected 2-D CGRAs are a sweet spot in the design space.[1] These CGRAs have been shown to provide 2-10X better power-performance than a traditional CPU [18, 19, 20], and while they have non-trivial challenges in placement and routing and desired optimization passes, we show these issues can be overcome. Our second insight is that the restricted scope of CGRA-targeted DBT allows it to have lower power and area overhead than general DBT and employ optimized versions of standard optimizations. Existing DBT and CGRA compilation work provides a good starting point and DORA incorporates new facets relative to each to create an effective CGRA DBT system. As a whole, this work makes the following contributions:

- It presents the first dynamic translation system for CGRAs that simultaneously achieves substantial energy and power benefits, matches the performance of offline compilation, and benefits the current execution,
- It develops approaches to existing optimizations that efficiently target a CGRA,
- It demonstrates how dynamic information can be captured and leveraged to improve the created translation,
- It describes an effective, low-overhead placement algorithm for 2-D CGRAs, and
- It quantitatively compares using an existing core and an ultra-low power microcontroller to perform the optimization and finds that the low power core is competitive and even advantageous to using an existing core.

2. CGRA REVIEW

Coarse Grain Reconfigurable Architectures are a broad class of architectures that aim to use a configurable datapath to achieve the performance of custom hardware with the flexibility to target many types of computations. They incorporate multiple functional units within a configurable interconnection network. By creating customized datapaths and eschewing power-hungry pipeline structures, they can achieve notable performance and energy efficiency gains.

CGRAs differ in their organization, computational capabilities and system integration. Some examples include customizable-instruction accelerators like CCA [15], CHARM [9], a composable accelerator for domain specific computations, and LIBRA [20], a configurable SIMD accelerator.

Fine grained reconfigurable architectures like FPGAs have similar benefits, but allow modifications at the bit level. This presents further optimization potential, as simpler datapaths can be utilized for computations requiring fewer bits. This comes at the cost of many more potential configurations, which can require more sophisticated compilation algorithms that generally run for prohibitively long durations. Also, it is usually extremely difficult or impossible to determine if bit-width modifications are permissible from the program binary alone. For these reasons, fine-grained architectures are less attractive targets for dynamic translation.

[1]Compared to the 1-D feed-forward CGRAs [21, 15, 16, 22] or fine-grained RAs [23] targeted by existing work.

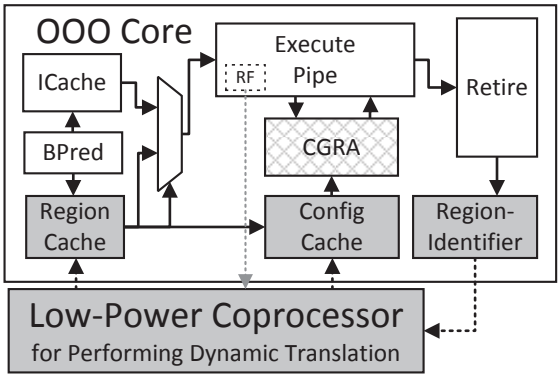

Figure 1: Overview of DORA transparent integration. Shaded blocks indicate new elements. (RF: Register File)

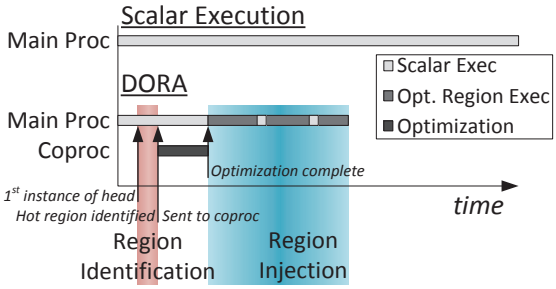

Figure 2: Timeline of DORA optimization process.

3. SOFTWARE-TRANSPARENT CGRA INTEGRATION

Traditionally, either new programming models and languages or specialized compilers with static analysis to identify candidate regions have been required to create CGRA-specific code. Dynamic translation has the potential to employ CGRAs without modifying the original application. To be viable, such a system should be integrated non-intrusively with the rest of the processor, should not degrade the performance of the rest of the system, and, ideally, should provide CGRA execution efficiency on par with static compilation.

Our approach (DORA), shown in Figure 1, has three main elements, similar to those in other HW DBT systems. The first, region identification, monitors retiring instructions to construct candidate regions and selects the most opportune for translation and optimization. Selected regions are sent to the second element, which translates the region for execution on the CGRA hardware. This produces configuration information for the array and, if appropriate, modified software for the supporting processor. This element also monitors register values for select instructions to facilitate optimizations based on runtime information. The third element, region injection, stores the generated software in a special region cache. Future invocations of the region execute from this cache. Figure 2 shows an abstract timeline of the region

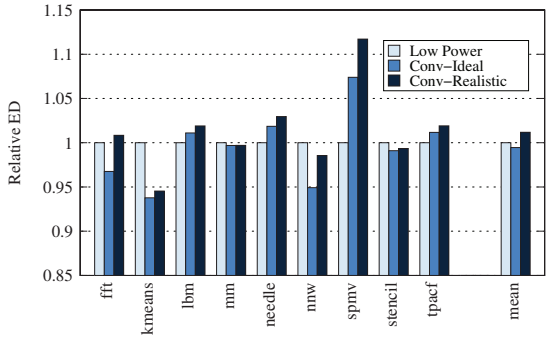

Figure 3: Whole program energy×delay for different translation processors relative to the low power microcontroller.

identification, optimization, and injection phases and how this compares to traditional scalar execution.

While there are multiple potential options to perform region translation, we focus on using a processor, either an existing core or a small, low-power processor integrated with the standard processor, as this minimizes design effort and facilitates future algorithm changes. The choice of using a dedicated versus existing processor is not obvious and, to our knowledge, has not been quantitatively compared. While an existing core leverages existing hardware and can provide better performance, a small microcontroller can be much lower power and have easy access to internal information on the main core. To guide our decision, we evaluated the energy-delay of running our translation algorithm (discussed in Section 4) on both our base out-of-order core and a very low power in-order core (details in Section 5). For running on the base core we considered two cases: 1) an ideal case where the translation thread runs concurrently with the main thread and there is no interference or communication overhead and 2) a case where only one of the two threads is running and there is minimal context switch overhead. Figure 3 shows the relative energy×delay of the three options. While the base core is better by 0.5% in an ideal case, the low-power microcontroller is a better trade-off when real constraints are considered (by more than 1% in this particular case). Further, a dedicated core can have direct access to the main processor, allowing easier access to runtime information. While adding a coprocessor does increase design and verification complexity, the overhead should be small as very low power processors, like the ARM M0 or Xtensa LX3, exist to leverage. Based on this analysis we use a small coprocessor to perform the translation.

The remainder of this section describes region identification and injection and how it could be integrated with various CGRAs. Section 4 then describes the translation and optimization process to target a particular CGRA.

3.1 Candidate Region Identification

Similar to other hardware DBT work, DORA monitors retiring instructions to find candidate regions for optimization. Given our target, we seek regions that are amenable to efficient CGRA execution. Specifically, the hardware attempts to find traces with high degrees of computation, especially tight loops, to leverage the CGRA's computation resources.

We employ a variant of the next-executing tail (NET) [24] algorithm to identify candidate regions. NET works to identify hot code segments by monitoring instructions which are the target of backwards branches (which often identify looping code). NET counts each time a potential trace head is executed. When a particular counter exceeds a threshold it records all retired instructions until a stop condition occurs. Stop conditions include an instruction that is a backwards branch, start of another trace, or exceeds the trace capacity.

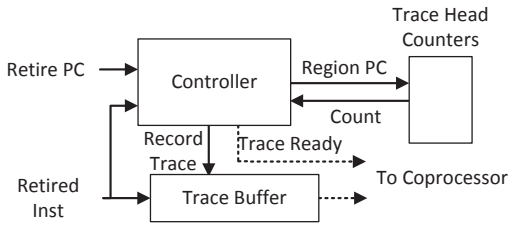

Figure 4: Region Identifier.

Figure 4 shows the region identification hardware. The controller monitors the retiring instruction stream for backwards branches. On finding a backwards branch, the region head counter table is queried and updated, either inserting the new head if absent or updating its count. While multiple options are available for the table, we found that a simple 32-entry direct-mapped cache addressed by the instruction PC works well in practice for our workloads, identifying all hot regions.

Candidate regions can vary widely in terms of their size. Since larger regions execute more instructions per instance, and therefore impact performance more, they can be considered to become "hot" more quickly. To account for this, the counter for larger regions is incremented more for each instance. When inserting a new head, the number of instructions executed until a stop condition is counted and a log approximation of the count (based on the most significant bit) is stored. This instruction count proxy is used on each subsequent occurrence to increment the counter.

When a counter exceeds a threshold, the trace buffer is signaled to record all retiring instructions until a stop condition occurs. While the trace is being created, the controller monitors the inserted instructions to ascertain if the trace is a good candidate for CGRA optimization. In particular, it tracks the number and types of computation and the amount and direction of control flow instructions. Regions that include computation instructions that exceeds a set threshold, or contain a tight loop (one that loops back to itself) with any computation and no internal control flow, are flagged for optimization.

3.2 Optimizing Regions

Once a trace is ready, an interrupt is sent to the coprocessor. The coprocessor uses an AXI interface [25] to read the contents of the trace buffer into local memory. Once the read is complete, the region identifier resumes collecting new traces. Section 4 describes the optimization process.

3.3 Optimized Region Injection

To make the optimized code available to the main processor a simple *region cache* is added to instruction fetch. The coprocessor uses an AXI interface to place the generated processor code into the region cache and the new configura-

Architecture	Targeted Accelerator	DORA Application
CCA [15]	Feed-forward configurable functional unit	Identify high compute to memory ratio basic blocks. Create CCA configuration for computation instructions and place code with invocation of CCA in region cache.
CHARM [9]	Shared heterogeneous Accelerator Building Blocks (ABBs)	Identify instructions that can be accelerated by ABBs. DORA coprocessor takes the place of accelerator block coordinator and coordinates use of shared ABBs. Selected instructions are replaced with utilization of ABB.
Libra [20] & DySER [18]	Configurable CGRA with Vectorization	Identify looping traces (and non-looping for DySER) with non-trivial computation. Perform optimizations and place in configuration cache. (No region cache needed for Libra as accelerator takes control in optimized code.)

Table 1: Application of DORA to different architectures.

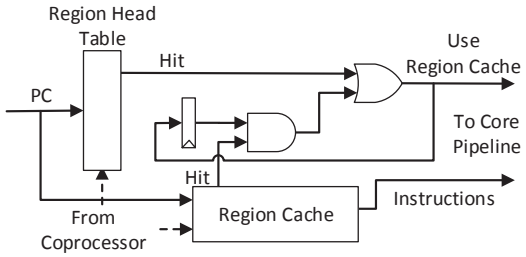

Figure 5: Region cache organization.

tion into the CGRA's configuration cache. The coprocessor directly manages both structures to ensure they stay in sync and all information for a given optimized region is available. If a region or configuration must be evicted due to lack of space, the coprocessor evicts the associated entries in the other structure as well. The region cache uses a simple 1-bit clock algorithm to aid the coprocessor in selecting a region to evict. Evicted entries could be stored in a lower memory level, but this could complicate the desired architectural transparency and is beyond the scope of this work.

The region cache, shown in Figure 5, contains two main pieces. A Region Head Table contains the PC of the first instruction of all optimized regions. The main Region Cache (a blocked array) contains the instructions for the optimized regions. Using a separate cache as opposed to part of the existing cache allows the coprocessor to have direct control of the contents. It has the side benefit of lower energy consumption when executing from the smaller cache.

To correctly direct execution, the region cache monitors the PC. When the current PC hits in the Head Table a flag is set to execute instructions out of the region cache. Execution continues from the region cache until a miss or direct jump occurs, at which point instruction fetch is directed back to the main instruction cache.

Like other DBT work, DORA must handle exceptional cases. A single bit of state is added for the register indicating the processor is executing from the region cache. On a precise exception which switches back into the same core, like a page fault, this bit will tell the processor which cache to resume from. To precisely debug arithmetic exceptions, like divide by zero, a processor debug mode can disable DORA. If in an optimized region, imprecise interrupts, like a timer interrupt, will wait until a branch and transfer to a special cleanup region to appropriately set the next PC. For simplicity, pages with DORA optimized regions are marked read-only. The fault handler will return to the original version of the code in the case of self-modifying code.

3.4 Transparently Integrating Various CGRAs

To show how this architecture could target different CGRAs, we show how DORA would be used to target CCA [15], CHARM [9], Libra [20], and DySER [18]. Table 1 details the nature of each accelerator and how DORA could target each architecture. CCA proposed their own dynamic translation option. They utilized the more hardware-intensive rePLay framework [26]. DORA can target CCA with much less overhead.

4. DYNAMIC TRANSLATION

Once a candidate region of code has been identified, it must be transformed to employ the CGRA. For our targeted architecture, this includes creating both configuration information for the CGRA hardware and software to interface with the array. The simplest approach, which is what has been employed in previous CGRA DBT attempts [14, 15], is to process each instruction in sequence and map any candidate instructions to the next available functional unit in the fabric. Especially for larger, more complex CGRAs, however, this can leave much of the available benefit unrealized.

DORA employs a series of optimizations both before and after actually mapping the computation to the CGRA to transform the original region into something that best utilizes the available computation resources. The relevant optimizations depend on the target architecture. We evaluated a number of potential optimizations from different sources. In the end we focused on those used by the custom compiler for our target architecture, DySER [11] (see Section 4.1), and those that leverage dynamic register content for specific instructions. When DORA targets a different architecture, different optimizations may be optimal. For example, many CGRAs benefit from modulo scheduling [27, 28]. DySER, however, does not require this optimization as it gets an implicit form of modulo scheduling due to its integration with an out-of-order core and its support for pipelined execution.

Table 2 lists the optimizations performed by DORA and classifies the optimizations that are also used in the DySER compiler [11]. A number of optimizations achieve similar results (marked with an 's') in both DORA and the compiler, although the underlying implementation is different due to the different initial sources. Others, in particular the CGRA mapping, have similar goals, but the approach used by DORA is different and simpler due to the need to perform

Optimization	Comp.	Trans. Time Dec.
Loop Unrolling	s	64%
Loop Store Forwarding	s	0%
Loop Deepening	s	5%
Ld/St Vectorization	s	9%
Accumulator Extraction	s	-4%
Dead Code Elimination	e	0%
Op Fusion	e	0%
Runtime Constant Insertion	n	0%
Dynamic Loop Transformation	n	0%
CGRA Placement	a	N/A

Table 2: Transformations performed by DORA, a classification if the compiler performs a similar (s), existing (e) or more advanced (a) version or does not (n) perform the transformation, and the average DORA translation time decrease when the optimization is removed.

the optimizations quickly. The compiler achieves a few optimizations with existing passes that DORA must perform explicitly. Other optimizations only DORA can achieve as they utilize runtime information. While many of the optimizations have been applied by DBT systems targeting other architectures, we are the first, to our knowledge, to rework them for CGRA-targeted DBT.

When selecting and implementing these optimizations we had to determine that they could feasibly be performed in a dynamic setting. This is a trade-off between additional translation time versus the performance benefit. The execution overhead of an optimization is influenced both by how long it takes to perform the optimization and how the optimization impacts the time consuming array mapping pass (Section 4.5). For many optimizations, the cost of performing the optimization is offset by decreased array mapping time. The last column of Table 2 shows the average decrease in translation time from removing the given optimization. In isolation, loop unrolling is by far the most expensive optimization. This is because it not only takes time to perform, but it often drastically increases the amount of computation to be mapped to the array. For some optimizations, such as accumulator extraction and store forwarding, the optimization actually decreases overall execution time as the decrease in mapping time more than offsets the optimization overhead.

Many optimizations synergistically impact performance when performed together. This is especially true for loop unrolling, loop deepening, and vectorization. For example, in STENCIL, loop unrolling or vectorization in isolation only provide 17% and 14% speedup, respectively. When combined the speedup increases to 62%. Adding loop deepening increases the speedup to 160%. These synergistic interactions make it difficult to isolate the benefit of individual optimizations as the full benefit is only realized when coupled with other optimizations.

While DORA faces many challenges not faced by compilers, it eliminates one of the major shortcomings of existing CGRA compilers. Existing compilers require either the programmer or offline profiling to identify regions to map to the array. DORA, in essence, profiles the application based on actual inputs and automatically adapts to changes in inputs.

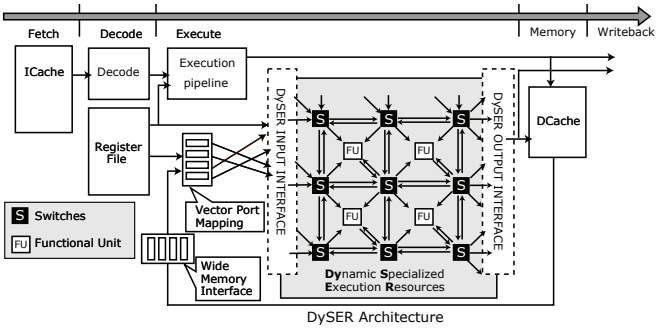

Figure 6: Overview of DySER architecture.

4.1 Target CGRA Review

To evaluate DORA we target a specific CGRA, DySER [18]. DySER is part of a subclass of CGRAs that integrate the array as a customizable function unit within the processor. DySER employs a sparsely connected 2-D array of semi-fixed function units with a configurable circuit-switched interconnect. As mentioned previously, we specifically target a 2-D array as we find it balances the possible performance and power benefits with achievable mapping complexity. Figure 6 shows an overview of DySER. A region optimized for DySER consists of two parts. The first is the bit stream that configures the DySER fabric. The second is a section of standard code, termed the *load slice* in DySER parlance, which sends data to and receives data from DySER either via registers or memory and handles the control flow and any computation not performed by the fabric. The load slice is the supporting software stored in the region cache described in Section 3.3. DySER's flexible I/O interface allows different portions of a wide data chunk to feed different ports or multiple pieces of data to feed a single data port in sequence (or a combination of both). DySER can also embed constants in the array.

4.2 KMeans Example

To facilitate better understanding of the translation process, we will walk through an example from KMEANS as different elements of the process are described. Figure 7(a) shows the original source code of the identified hot region.

4.3 Pre-Mapping Processing

When the coprocessor receives an interrupt indicating a new trace is available, it reads the contents of the trace buffer into local memory. In many cases, the trace is a single looping basic block. In cases where the trace contains multiple basic blocks, the translator extracts the most promising block based on the number of computation instructions. The rest of the translation process is performed on the selected basic block. Future work will look at the possibility of if-conversion [29] or speculation to create larger superblocks.

The instructions are analyzed to determine dependency information and make a first pass at selecting instructions for CGRA optimization. Load, store, control flow, compare, and address generating instructions are all excluded from placement in the array. Address generating instructions are identified by back-propagation from the address registers of memory operations. Details on the handling of compare in-

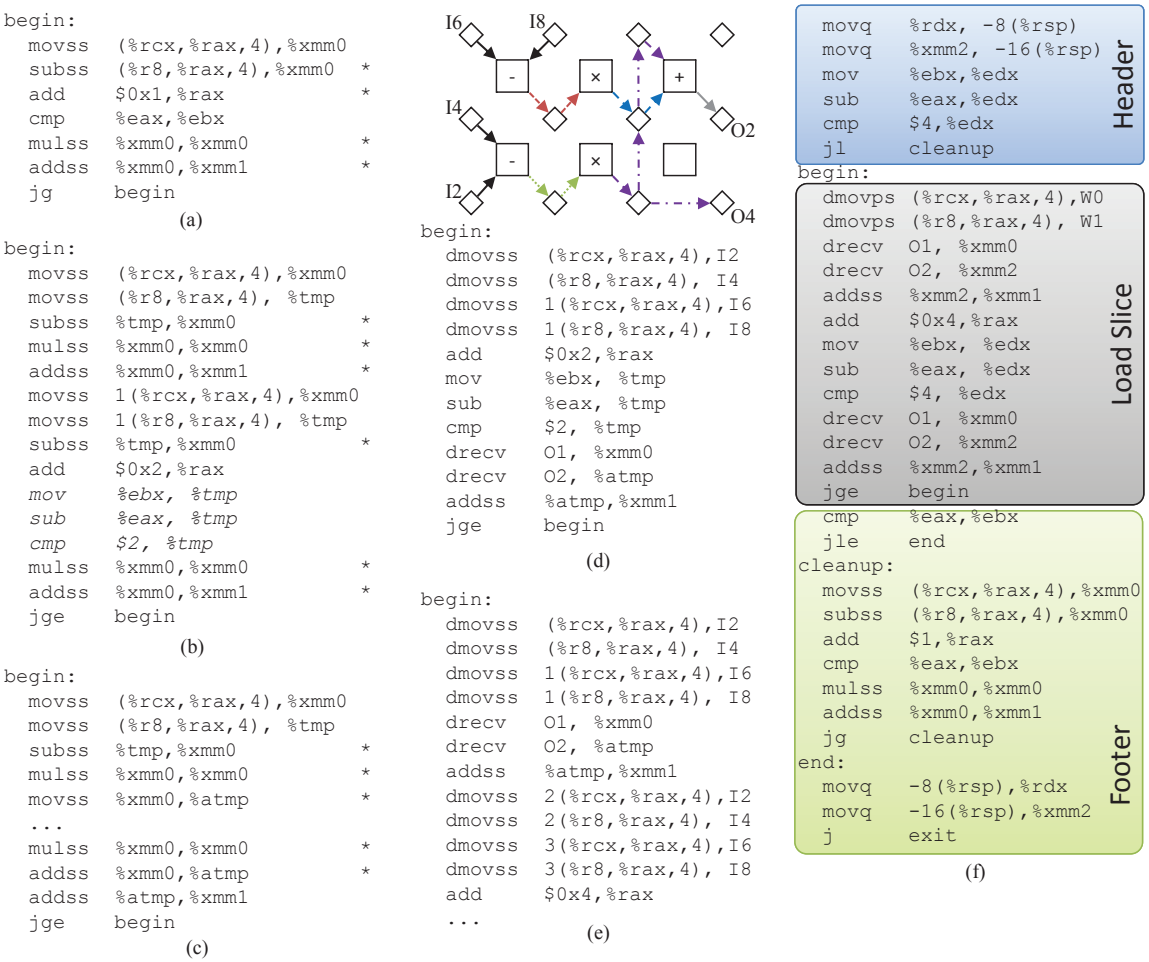

Figure 7: Example progression of dynamic translation on KMEANS loop. The original source code (a), the unrolled loop (b), accumulator conversion (c), generated scheduling and load slice (d), result of loop deepening (e), and final code region (f).

structions are discussed in Section 4.5. Complex mem-op instructions are broken apart into two separate instructions to facilitate inclusion of the operation portion in the array. The *'s in Figure 7(a) indicate the instructions initially identified for mapping. After this initial analysis, a number of transformations are potentially performed on the region to improve the benefit it receives from the reconfigurable array.

Induction Register Identification and Exclusion.

An induction variable (or register) is one that is only updated by a fixed amount each loop iteration. They are typically used to monitor loop progress and to sequence through memory locations and are best executed outside the CGRA. Induction variable identification looks for instructions that are only updated by constant amounts and are only used by memory and control flow instructions. The mapping flag of identified instructions is cleared. In the example, this deselects the `add $0x1, %rax` instruction.

Loop Unrolling.

Many identified regions are loops with computation that does not fully utilize the computation ability of DySER. Loop unrolling aims to increase the amount of computation performed in a single pass. The amount to unroll the loop is based on the instruction types in the region and the available computation resources in the array. In our example, the original region contains two mappable FP add/subtract operations and one FP multiply. If our example array contains four FP add and four FP multiply units, the region could be unrolled once, resulting in two total iterations.

If unrolling is possible, the loop instructions are appropriately cloned and updated. The result of these transformations is shown in Figure 7(b). In many cases it cannot be determined from the instructions how many times the loop will iterate. To ensure correctness, the loop comparison must be modified to make sure there are at least as many iterations remaining as times the loop is unrolled. Figure 7(b) shows the modified comparison in italics. To handle any remaining iterations, the original version of the loop is added to a clean-up section discussed further in Section 4.6.

Accumulator Identification.

Accumulation is a common operation that introduces a register dependency that prevents pipelining iterations in the array. The solution is to have the array output the sum of just the internal values and add this to the overall sum in software. DORA identifies accumulator registers by looking for registers that are both the source and destination of

add instructions and are not used by any other instructions. Identified instructions are transformed to accumulate into a temporary register and an unmapped instruction is inserted to add the temporary result to the accumulator register. In our example, register `%xmm1` is identified as an accumulator. The result of the accumulator transformation is shown in Figure 7(c).

Store Forwarding.

Some loops store a value that is loaded in subsequent iterations. When unrolled, this leads to a dependency where the stored value is first output from and later loaded back into the array, degrading performance. Store forwarding identifies store-load pairs where the address calculation for both operations is the same, the address registers have not changed, and there are no intervening stores. In these circumstances the load is converted to a use of the stored register value. This results in implicit data forwarding within the array.

4.4 Runtime Register Monitoring

By monitoring the register values of certain instructions DORA is able to perform dynamic optimizations unavailable to the compiler. DORA monitors the value of registers for particular instructions to identify runtime constant registers and direct the applied optimizations, in particular the amount of loop unrolling and deepening. To select relevant registers to monitor, the initial pre-mapping pass identifies computation instructions with registers that are read but never written within the region and registers involved in loop comparisons. Monitoring hardware is setup to observe the register values for these instructions while the rest of the pre-mapping optimizations are performed. This allows the register monitoring to overlap with other useful work so as not to delay the optimization process. The pre-mapping stage is long enough (100,000s of cycles) to obtain a good sample, but short enough to add negligible energy overhead for the register monitoring.

Monitoring Hardware.

The monitoring hardware monitors a register value for a particular instruction and identifies when the value differs from the previous execution. To achieve this, a small set of registers in the decode stage hold the instruction PC and register of interest. When an matching instruction is decoded, a flag is set to monitor the read register. When the identified register is read, the value is compared to the last value of the register for that instruction. If the value is different, an interrupt is sent to the coprocessor to read the new value. The register can alternatively be set to raise an interrupt when the value is less than the currently stored value. The latter is useful when inspecting loop comparisons based on memory addresses. In our implementation, we support monitoring up to six different instructions simultaneously. Each instruction monitor requires 66 bits (64 for the PC, one to select which of the two register values to monitor, and a valid bit) plus a comparator in decode and a 32-bit register, a comparator, and logic to generate a coprocessor interrupt in register read. No additional register ports are required as register information is already available in each relevant stage.

Dynamic Register Optimizations.

One use of register monitoring is to detect runtime constants. If the value of a locally read-only register remains constant for many iterations it often remains constant for the entire run. An identified register constant can be embedded in the CGRA mapping, obviating the register read and routing through the array, which saves power and potentially improves performance. To be safe, a check is added to the loop header to confirm that the register matches the expected constant in later iterations.

The second use is to monitor the number of dynamic iterations of a register bound loop to guide the amount of unrolling. If a non-trivial number of iterations have a small bound, as happens in FFT, unrolling can actually prevent many iterations from utilizing the optimized region. If the monitoring finds iteration limits that suggest that more than a threshold number of iterations will execute the unoptimized code if unrolled, then the code is reverted to the pre-unrolled state and the remaining pre-loop optimizations are reapplied. This allows a deepened version of the loop (Section 4.6) to be used in the main body and a partially optimized version using the same configuration to be used in the cleanup code, providing benefit to all iteration bounds.

In all cases, register monitoring is ended as soon as possible to save energy. For constant identification, for example, the monitoring is disabled if a register is seen to have more than a single value. In our example, registers `eax` and `ebx` of the compare are monitored. The `ebx` value ends up being constant and large enough that loop unrolling is kept.

4.5 Dynamic Mapping

Once the computation has been transformed into a form most suitable for DySER, the computation must be scheduled on the array. Scheduling for spatial architectures is typically NP-complete. CGRA compilers generally utilize integer linear programming, satisfiability modulo theory, or some architecture-specific polynomial-time approximation. The former two are too time consuming for our dynamic environment so we use a version of the latter.

We develop a greedy algorithm that sequences through the transformed candidate instructions in program order. For each instruction, it selects a function unit closest to the mid point between the input values to the operation and then attempts to find a viable routing from the source locations of the inputs to the function unit. It continues trying nearby function units until either a successful placement is found or all potential function units have been explored. Scheduling stops if a) an appropriate function unit is not available, b) data cannot be routed to any available function unit, or c) the end of the code region is reached. In either of the first two cases, any remaining instructions originally slated to execute on the array are marked to instead be executed as standard instructions in the load slice.

When a needed source value is not present in the forming DySER configuration, an instruction is added to the load slice to send the data to the array. Similarly, instructions are added to receive data from the array when computation is complete.

Routing is often one of the most expensive operations for reconfigurable arrays. In order to limit execution time, our

algorithm only explores routing options within a bounding box between the source and destination. While this could mean that routing fails even though a legitimate option exists, in practice we find this is rarely the case.

Once the scheduling is complete, the generated function unit configuration and routing paths have a direct translation to the array configuration bits. Both the generated load slice and DySER array mapping for our example are shown in Figure 7(d).

ISA Specifics.

X86 control flow and conditional instructions execute based on the EFLAGS register set by compare instructions. Since DySER does not have a flags register, DORA tracks the sources used by the most recent compare instruction. Conditional operations, like `cmovl` (conditional move on less than), scheduled to the fabric use these stored sources to appropriately schedule the instruction.

4.6 Post-Mapping Optimizations

After array scheduling is complete, additional optimizations can be made to the supporting load slice. These optimizations aim to increase parallelism and eliminate unnecessary code.

Loop Deepening.

Loop unrolling (Section 4.3) aimed to make full use of the computation resources available in the array. Loop deepening aims to make full use of the wide input and output operations available in DySER. Loop deepening analyzes groups of related loads and stores (those that differ only by their offsets). If the current groups do not utilize the full width of the available vector operations, then the loop code is replicated as many times as necessary to utilize the full width.

In our example there are two sets of related loads, those with base addresses `(%rcx,%rax,4)` and `(%r8,%rcx,4)`. Each group initially spans 64-bits. If the CGRA supports 128-bit wide loads, the loop can be deepened once to provide four 32-bit loads to the same address group. The result is shown in Figure 7(e).

Load/Store Vectorization.

Vectorization combines load and store operations that differ only by their offsets into single wide memory operations. Groups of related loads and stores are candidates for vectorization as long as it can be determined that they do not read or write a location that was written or read by an intervening memory operation. Cases, like our example, where there are only loads (or stores) in a region, or where the loads and stores are not interleaved, can always be vectorized. Figure 7(f) shows the packed load `dmovps` operations that result from vectorization in our example.

In cases where loads and stores are interleaved, more care is required. The translator attempts to add memory disambiguation checks to the header of the loop to ensure that the accessed regions do not overlap. It can add these checks, and therefore vectorize related loads and stores, when the address calculation registers for relevant loads/stores do not change or change only by a fixed amount. In such cases, a comparison is added to the header comparing the two base addresses. If they are equal, the optimized region is skipped and all execution happens in the cleanup section.

Other Optimizations.

A few smaller optimizations further cleanup the load slice. Dead code elimination removes any instructions whose output is not used before the register is written again. This can result when computation is performed within the array. An operation fusion pass combines safe mem-op pairs.

Clean-Up Code.

Transformed loops must only execute as many iterations as the original version. This often requires additional code to ensure proper execution. For a register bounded loop, such as our running example, a header is needed to ensure the loop will execute at least as many iterations as performed by the optimized region, and a footer is needed to complete any additional iterations. In a constant bounded loop the header can be omitted and the footer may not be needed depending on the specific bound. Figure 7(f) shows the header and footer for the `kmeans` example. The header checks that the loop will be executed at least four times. The footer first checks if there are any iterations remaining, and if there are, executes them using the original region code.

In cases where temporary variables are needed, the header and footer include stack operations to save the values used as temporaries. The selected temporary variables are registers that are not used in the optimized region. Finally, since the number of instructions in the final code is unlikely to exactly equal the number in the original region, a jump to the first instruction after the optimized region is added.

4.7 Implementation

Our C++ implementation of the described algorithm consumes less than 4000 lines of commented code. We model the coprocessor after the Xtensa LX3 [30]. The LX3 achieves a 1 GHz clock while occupying 0.044 mm^2 for the core alone and consuming 14 mW at 45nm, providing both speed and low power. The coprocessor has 32kB data and 16kB instruction memories. Simulation with these caches shows miss rates less than 1%, indicating the coprocessor can execute out of memories this size.

5. EVALUATION METHODOLOGY

We compare the performance of DORA to the performance of hand and compiler optimized DySER execution.

5.1 Modeling

We use gem5 [31] to model performance and McPAT [32] and Cacti [33] to model power. The main core is a 4-way out-of-order x86 processor with 64 kB L1D and 32kB L1I caches and a tournament branch predictor. The main core and DySER fabric run at 2 GHz. Since gem5 does not support the Xtensa ISA, the translation microcontroller is modeled in gem5 as a single-issue in-order ARM core executing the embedded Thumb ISA running at 1 GHz. Area estimates based on McPAT and available literature at the 55/45nm technology node for the major components are shown in Table 3. The 0.32 mm^2 added for DORA is negligible compared to the rest of the core.

4w-OoO	53.5 mm^2	32-E DM Trace Head	10.9 μm^2
Coproc	0.17 mm^2	16-E DM Region Head	7.4 μm^2
Region Cache (4kB)		0.15 mm^2	

Table 3: Component area estimates (E=entry).

	# Reg	LU	ACC	SF	LD	Vect	RRI
fft	1	x			x	x	x
kmeans	2	x	x		x	x	
lbm	1					x	x
mm	1	x	x		x	x	
needle	1	x		x		x	x
nnw	3	x	x		x	x	
spmv	1	x					
stencil	1	x			x	x	x
tpacf	1					x	x

Table 4: Evaluated benchmarks (LU=Loop Unrolling, ACC=Accumulator Identification, SF=Store Forwarding, LD=Loop Deepening, Vect=Ld/St Vectorization, RRI=Runtime Register Information).

Similar to Govindaraju et al. [11], we consider a heterogeneous DySER array with 16 INT-ADD, 16 FP-ADD, 12 INT-MUL, 12 FP-MUL, 4 FP-DIV, and 4 FP-SQRT units. In their area analysis they state that this configuration has the same area as an AVX unit and twice that of an SSE unit.

5.2 Benchmarks

We evaluate DORA using the Parboil benchmark suite [34]. We select this suite for two reasons. One, the benchmarks present a challenging but high potential set of workloads. The scalar code is written without a particular target in mind, but the workloads contain enough data parallelism to be good candidates for acceleration. Second, they are the workloads evaluated by the released DySER compiler [11], providing access to both compiler and manually optimized versions.[2] Table 4 lists the benchmarks we evaluate. The table also lists the number of regions DORA identifies and then selects for optimization as well as the primary transformations that each workload benefits from. With two exceptions for the compiler, the hand and compiler versions optimize the same regions as DORA.

6. RESULTS

We wish to determine i) how close DORA performance comes to compiler (and manually) optimized code, ii) if the identification and optimization overheads are small enough when performed on a low power core to not outweigh the benefits, and iii) how DORA+DySER compares to SIMD execution.

6.1 DORA Performance

First, we look at the quality of the configurations created by DORA irrespective of the time to create the configuration. Figure 8 shows the speedup of manually, compiler, and DORA optimized code relative to the original scalar version.

[2]The Parboil version used by [11], and therefore the one we use, is an earlier version with a slightly different mix of benchmarks than the current release.

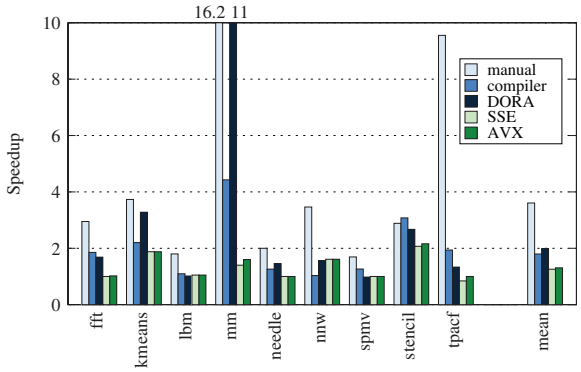

Figure 8: Whole program speedup of DySER mapping techniques and compiler SIMD extensions relative to scalar code.

As would be expected, manually generated mappings provide the most benefit at 3.6X speedup. The compiled code provides 1.8X speedup while DORA achieves an average speedup of 1.99X. Not only does DORA match the performance of the compiled code, in many cases, it actually exceeds it. There are three reasons for this. First, the exact way optimizations are applied by the compiler and DORA differ in some cases. Second, in two instances, DORA chooses to optimize regions the programmer-guided compiler does not. Finally, DORA's runtime register monitoring allows optimizations not available to the compiler.

The performance of the different benchmarks can be roughly categorized into four groups.

Manual Equivalent Performance.

Three workloads, KMEANS, MM, and STENCIL achieve performance similar to the manually optimized versions. Looking into the configurations generated, DORA creates configurations identical or nearly identical to the manual versions. For two of the workloads, KMEANS and MM, DORA achieves better performance than the compiler. MM is impacted by the vectorization optimization selected. Specifically, DORA vectorizes by unrolling and reducing while the compiler uses scalar expansion (which cuts the reduction dependence). In this case, unrolling+reduction is better because it reduces L1 cache bandwidth use, which is critical for MM. In KMEANS, DORA identifies a second region to optimize which is not identified by the programmer-directed compiler.

Compiler Equivalent Performance.

FFT, LBM, NEEDLE, and NNW match or exceed the performance of the compiler generated code, but fall short of that achieved by manual optimization. The number of iterations of the major inner loop of FFT varies during execution, including many small values. While the runtime register analysis guides DORA to not unroll the loop, this is less optimal than the per-iteration count optimization of the hand version. NNW contains a constant memory lookup table that neither the compiler nor DORA can reason about to fully vectorize the lookups. NEEDLE contains an inter-loop dependency that limits parallelism. In LBM, neither the compiler nor DORA achieve much benefit. LBM has a very long hot region and currently only a single DySER function

is created. The manual version achieves better, but still limited, speedup by optimizing multiple sections of the region.

Suboptimal Benefit.

TPACF achieves speedups less than the compiler. TPACF faces two issues. The compiler has programmer help to specify a specific region of the large hot loop to optimize. This region exposes more parallelism than optimizing from the head as DORA does, leading to larger gains. Second, the scalar loop header loads values into registers for use within the loop. While this is advantageous for normal execution, it prevents vector loading this information to the array.

Minimal Benefit.

SPMV sees no performance improvement with DORA. This benchmark is a challenge in general, with manual and compiler optimizations achieving only 69% and 26% speedup, respectively. DORA cannot dynamically guarantee that the load and store addresses accessed by the configuration are distinct and so vectorization is not possible. The compiler version side steps this by modifying the code to explicitly guarantee there is no aliasing.

6.1.1 Runtime Register Monitoring

DORA's ability to monitor the runtime value of registers for particular instructions provides substantial benefits for two benchmarks. By monitoring the loop comparison registers in FFT, DORA is able to determine it is better not to unroll the loop so that all iterations can utilize a version of the DySER function, increasing the benefit provided by DORA for FFT by 80%. For NEEDLE, DORA embeds runtime constants in the DySER function, yielding a 37% improvement in DORA's performance.

6.2 Dynamic Optimization Overhead

Even if DORA produces perfect implementations, they will only be useful if they are available in time to be used. To evaluate this, the translation algorithm is simulated on our coprocessor model for each hot region. From this we determine how long translation takes, how it compares to the overall execution time, and how this delay impacts the achieved speedup.

Region identification happens quickly, with all but one hot region being selected less than 8000 cycles after it first executes. Region optimization and placement is more costly. Table 5 shows the amount of time to translate each region and how this compares to the total scalar execution time. On average, translation takes 3.8 ms. For comparison, the DySER related compiler passes take an average 1530 ms to perform[3] (a 400X difference).

Table 5 also shows the performance lost when including translation overhead compared to zero-overhead dynamic translation. Even with translation equaling up to 27% of scalar execution time, realistic DORA is still within 16% of the ideal in all cases and within 5.1% on average. Realistic DBT achieves a 1.88X speedup compared to 1.99X with zero-overhead optimization, showing that the translation overhead is small even for relatively small input sets and still better than the 1.8X of compiled code.

[3]Pass overhead determined from the compiler's timing report.

	Time (ms)	% Scalar Exec	Perf Loss
fft	4.7	27.0%	15.6%
kmeans R1	1.6	3.2%	10.0%
kmeans R2	1.1	1.6%	
lbm	3.3	5.1%	0.1%
mm	1.8	0.0%	0.4 %
needle	4.5	6.9%	3.0%
nnw R1	1.7	4.8%	12.1%
nnw R2	5.0	13.7%	
nnw R3	2.1	5.9%	
spmv	2.2	24.8%	0.0%
stencil	1.7	1.5%	2.3%
tpacf	15.1	5.0%	1.7%
Mean	3.8	4.0%	5.1%

Table 5: Cycles to optimize each region on the coprocessor, percentage of scalar execution time, and performance degradation relative to zero-overhead dynamic translation.

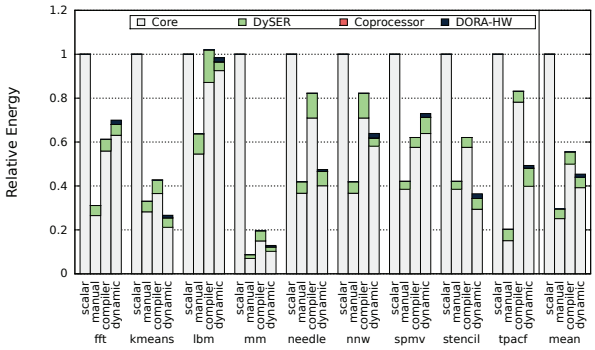

Figure 9: Energy breakdown of DySER mapping techniques relative to scalar code.

As further mitigation, the input sets provided as part of the Parboil suite are intentionally kept small to keep simulation times short. More realistic inputs would result in longer overall execution times and even smaller relative overheads. Further, while efficiency was considered when designing the translation code, our first goal was flexibility. A refined version focusing on speed could reduce the translation execution time.

6.3 Energy Analysis

Adding hardware for DORA consumes power. Since one of the major advantages of CGRAs are their energy efficiency, the additional energy for DBT should be small so as not to overly diminish the energy savings. Figure 9 shows the relative energy consumption for all four options normalized to the scalar baseline. The figure shows the energy consumed by the main core, the DySER unit, and, in the case of DORA, the coprocessor and the identification and insertion hardware. When not optimizing a region, the coprocessor is assumed to be clock gated, waiting for an interrupt.

The manually and compiler optimized versions consume 30% and 57% of the baseline energy on average. DORA consumes 47% of the baseline energy. Even with the added overhead for dynamic translation, DORA consumes less energy than the compiler in 77% of the cases. In some cases

	Year	Trans. Target	Optimization Details	Program Perf. Impr.	Match Offline	Trans. Time
DIF [35]	1997	VLIW Engine	Speculation; LIW creation	small	NR	low
DAISY [36]	1997	VLIW	StdOpts, loop unroll	NR		low
Transmeta [37]	2000	VLIW	StdOpts, loop unroll	NR	NR	NR
Yehia & Temam [22]	2004	FG Func Unit	noFP; collapse comb. logic	small	NR	est. low
CCA [15]	2004	1-D FF CGRA	noFP; no optimizations	small	NR	est. low
Warp Processor [23]	2004	FG FPGA	noFP; loop rerolling, strength prom.	large	x	high
DIM [38]	2008	1-D FF CGRA	BB+speculation; noFP	small	NR	est. low
GAP [39, 17]	2010	1-D FF CGRA	BB+speculation; noFP	small	NR	est. low
Ferreira et al. [27]	2014	Xbar CGRA	Modulo scheduling; noFP; VLIW src	NR	x	low
DynaSPAM [16]	2015	1-D FF CGRA	OoO hardware for scheduling	small	NR	low
DORA	2016	2-D CGRA	Vectorization, Loop unroll & deepen	large	x	low

Table 6: Overview of past data-parallel dynamic translation proposals, including whole program performance benefit (small = $\leq$42% improvement; large = $\geq$90% improvement), ability to match offline translation, and translation time (low = <10ms; high = >200ms). (BB: Basic block, FG: Fine-grained; FF: Feed-forward; StdOpts: Std DBT Optimizations; noFP: no floating point; NR: not reported)

energy consumption is better even though performance is not. This is due to the fact that DORA executes optimized regions out of its small region cache, which consumes less energy than running out of the main L1I cache.

Relative to the ideal energy consumption that DORA could achieve if the supporting hardware consumed no power, the additional hardware only increases average energy consumption by 2.8%. Almost all of this is from the region cache.

When DORA does not identify candidate regions, its overhead is minimal. Performance is not impacted and the maximum identification hardware power amounts to only 0.1% of the average main core power. In a pathological case where the coprocessor is constantly, but unsuccessfully, optimizing, the overhead increases to only 0.23% of main core power.

6.4 SIMD Compilation

SIMD units and CGRAs address a partially overlapping optimization target. Despite decades of work, automatic utilization of SIMD extensions, either dynamically or by compiler, has met limited success. While PARROT [21] performed a limited form of purely dynamic SIMDization, most wide ranging dynamic SIMDization requires an offline analysis pass [40, 41, 42, 43]. The last two bars of Figure 8 show the performance of code compiled for SSE and AVX extensions. In all cases, DORA performs as well or better than both extensions without requiring recompilation. While DORA's performance clearly outstrips compiled SIMD, SIMD's 30% benefit is similar to previous CGRA DBT, showing the importance of target CGRA and optimization selection.

7. RELATED WORK

DBT has a rich history and has been targeted at a variety of architectures. Many works optimize a set of instructions for execution on a typical processor, many translating to a different ISA at the same time [21, 44, 45, 46, 47, 7, 48, 26]. Focusing on those that perform translation in hardware, PARROT [21] and rePLay [26] identify hot regions of single-entry, single-exit blocks in hardware and perform various optimizations, including partial renaming, dead code elimination, and instruction fusion, on the regions. I-COP [47] introduces using a separate coprocessor to create and optimize traces for a processor's trace cache.

7.1 DBT for Data-Parallel Architectures

Of particular relevance to this work are proposals which translate code to employ a more power-efficient computing architecture. Existing works have proposed DBT for SIMD [21], VLIW [36, 37, 35], 1-D feed-forward or fully connected CGRA [38, 15, 27, 16, 39, 17, 22], and FPGA [23] architectures. Table 6 summarizes this past work. DIF [35], DAISY [36] and Transmeta [37] translate standard RISC or CISA ISAs to VLIW architectures. Warp Processing [23] proposes an architecture to dynamically identify and map code to a simplified FPGA. Due to the nature of the targeted FPGA and the complexity of the mapping process, they target only tight integer loops. Translation takes on the order of seconds, often relegating the use of the optimized region to future executions. Yehia and Temam [22] employ rePLay to target a fine-grained, look-up table-based customizable function unit. CCA [15], DIM [38, 14], and GAP [39, 17] dynamically translate to 1-D, feed-forward arrays. The reported performance benefit for reasonable array configurations is <30% for all three designs. Ferreira et al. [27] perform dynamic modulo scheduling from a VLIW ISA to a fully connected CGRA with internal storage. DynaSPAM [16] uses existing out-of-order hardware to schedule to a 1-D, feed-forward CGRA. While they briefly discuss targeting a 2-D array, it is not evaluated and would create suboptimal mappings due to the restricted nature of their scheduling frontiers. DynaSPAM's optimizations come implicitly through out-of-order scheduling and are influenced by the number of branches supported in a trace and the size of the instruction window instead of the target array itself. DORA, instead, performs optimizations tailored to the targeted CGRA. None of the above previous works achieve both large power and performance and low translation time.

7.2 Compiler-Supported DBT

Other works investigate compiler-supported DBT for power-efficient architectures. CCA [15, 49] also considered

having the compiler identify and rework candidate regions to allow efficient run time translation. VEAL [2] is a loop accelerator where the compiler identifies candidate regions and expresses them in a modulo schedulable form. A dynamic translator attempts to map the identified regions to the available hardware. HASTE [50] includes a feed-forward reconfigurable function unit in an embedded processor. Potential kernels are marked in the binary and mapped to the reconfigurable unit at runtime.

8. CONCLUSION

We propose DORA, the first software-transparent dynamic translation scheme for CGRAs which simultaneously achieves substantial performance benefits, low translation overhead, and benefits on par with offline mapping. When targeted at DySER, our system performs as well as or better than a compiler in nearly 80% of cases and provides 1.99X speedup over scalar execution without any modifications to the code. The translation overhead is minimal at a 5.1% performance and 2.8% power loss.

A major challenge to the adoption of semi-specialized architectures like SIMD and CGRAs is that they generally require software reengineering. After decades of work on SIMD compilation, SIMD autovectorization is hard, and only getting harder with successive generations like AVX. We show that dynamic CGRA translation performance, when targeted at the right architecture, can exceed compiled SIMD code and match compiled CGRA code, offering a replacement for both. Especially for environments where architectural transparency is critical, our results show that dynamic, transparent CGRA translation is feasible and is a favorable option compared to extending existing SIMD ISAs.

9. ACKNOWLEDGMENTS

We thank Karu Sankaralingam for his feedback as this work developed and for comments on draft versions of the paper. We also thank David Albonesi and the anonymous reviewers for their feedback on the paper.

10. REFERENCES

[1] D. J. Kuck and R. A. Stokes, "The burroughs scientific processor (bsp)," *IEEE Transactions on Computers*, vol. 31, pp. 363–376, May 1982.

[2] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pp. 389–400, IEEE Computer Society, 2008.

[3] M. Duric, M. Stanic, I. Ratkovic, O. Palomar, O. Unsal, A. Cristal, M. Valero, and A. Smith, "Imposing Coarse-Grained Reconfiguration to General Purpose Processors," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2015.

[4] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 370–380, ACM, 2009.

[5] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP)," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pp. 141–150, Dec 2003.

[6] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: Evaluating spatial computation for whole program execution," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pp. 163–174, ACM, 2006.

[7] A. Deb, J. M. Codina, and A. González, "Softhv: A hw/sw co-designed processor with horizontal and vertical fusion," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pp. 1:1–1:10, ACM, 2011.

[8] Y. Park, J. Park, and S. Mahlke, "Efficient performance scaling of future CGRAs for mobile applications," in *2012 International Conference on Field-Programmable Technology (FPT)*, pp. 335–342, Dec. 2012.

[9] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pp. 379–384, ACM, 2012.

[10] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pp. 305–310, IEEE Press, 2013.

[11] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 341–352, IEEE Press, 2013.

[12] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 12–23, ACM, 2011.

[13] M. A. Watkins and D. H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 497–508, IEEE Computer Society, 2010.

[14] A. C. Beck and L. Carro, *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques - Automatic Acceleration*. Springer, 2010.

[15] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pp. 30–40, IEEE Computer Society, 2004.

[16] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "DynaSpAM: Dynamic Spatial Architecture Mapping Using out of Order Instruction Schedules," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 541–553, ACM, 2015.

[17] S. Uhrig, R. Jahr, and T. Ungerer, "Advanced architecture optimisation and performance analysis of a reconfigurable grid ALU processor," *IET Computers Digital Techniques*, vol. 6, pp. 334–341, Sept. 2012.

[18] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 503–514, 2011.

[19] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1284–1291, ACM, 2012.

[20] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 84–95, IEEE Computer Society, 2012.

[21] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak, "Specialized dynamic optimizations for high-performance energy-efficient microarchitecture," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pp. 137–148, IEEE Computer

Society, 2004.

[22] S. Yehia and O. Temam, "From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pp. 238–249, IEEE Computer Society, 2004.

[23] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," in *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, (New York, NY, USA), pp. 659–681, ACM, 2004.

[24] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," *SIGPLAN Not.*, vol. 35, pp. 202–211, Nov. 2000.

[25] ARM, *AMBA AXI and ACE Protocol Specification*, 2013.

[26] S. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, June 2001.

[27] R. Ferreira, W. Denver, M. Pereira, J. Quadros, L. Carro, and S. Wong, "A run-time modulo scheduling by using a binary translation mechanism," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 75–82, July 2014.

[28] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 166–176, ACM, 2008.

[29] K. M. Hazelwood and T. M. Conte, "A lightweight algorithm for dynamic if-conversion during dynamic optimization," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pp. 71–, IEEE Computer Society, 2000.

[30] Tensilica, *Xtensa LX3 Customizable DPU*, November 2009.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 469–480, ACM, 2009.

[33] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi, "Cacti 5.1," Tech. Rep. HPL-2008-20, HP Labs, 2008.

[34] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep. IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[35] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pp. 13–25, ACM, 1997.

[36] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pp. 26–37, ACM, 1997.

[37] A. Klaiber, "The Technology Behind Crusoe Processors," tech. rep., Transmeta Corporation, Jan. 2000.

[38] A. Beck, M. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 1208–1213, Mar. 2008.

[39] S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer, "The Two-dimensional Superscalar GAP Processor Architecture," *International Journal on Advances in Systems and Measurements*, vol. 3, no. 1-2, pp. 71–81, 2010.

[40] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping," in *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, pp. 216–227, Feb. 2007.

[41] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor SIMD: Auto-vectorize once, run everywhere," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pp. 151–160, IEEE Computer Society, 2011.

[42] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 547–557, IEEE Computer Society, 2010.

[43] E. Yardimci and M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pp. 127–138, ACM, 2006.

[44] E. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer*, vol. 33, no. 3, pp. 40–45, 2000.

[45] E. R. Altman, K. Ebcioglu, M. Gschwind, and S. Sathaye, "Advances and future challenges in binary translation and optimization," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1710–1722, 2001.

[46] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pp. 1–12, ACM, 2000.

[47] Y. Chou and J. P. Shen, "Instruction path coprocessors," *SIGARCH Computer Architecture News*, vol. 28, pp. 270–281, May 2000.

[48] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta, "Performance characterization of a hardware mechanism for dynamic optimization," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pp. 16–27, IEEE Computer Society, 2001.

[49] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pp. 272–283, IEEE Computer Society, 2005.

[50] B. A. Levine and H. H. Schmit, "Efficient application representation for haste: Hybrid architectures with a single, transformable executable," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '03, pp. 101–110, IEEE Computer Society, 2003.