Exploiting Narrow Accelerators with Data-Centric Subgraph Mapping

Amir Hormati, Nathan Clark, and Scott Mahlke

Advanced Computer Architecture Lab
University of Michigan - Ann Arbor
E-mail: {hormati, ntclark, mahlke}@umich.edu

Abstract

The demand for high performance has driven acyclic computation accelerators into extensive use in modern embedded and desktop architectures. Accelerators that are ideal from a software perspective, are difficult or impossible to integrate in many modern architectures, though, due to area and timing requirements. This reality is coupled with the observation that many application domains under-utilize accelerator hardware, because of the narrow data they operate on and the nature of their computation.

In this work, we take advantage of these facts to design accelerators capable of executing in modern architectures by narrowing datapath width and reducing interconnect. Novel compiler techniques are developed in order to generate high-quality code for the reduced-cost accelerators and prevent performance loss to the extent possible. First, data width profiling is used to statistically determine how wide program data will be at run time. This information is used by the subgraph mapping algorithm to optimally select subgraphs for execution on targeted narrow accelerators. Overall, our data-centric compilation techniques achieve on average 6.5%, and up to 12%, speed up over previous subgraph mapping algorithms for 8-bit accelerators. We also show that, with appropriate compiler support, the increase in the total number of execution cycles in reduced-interconnect accelerators is less than 1% of the fully-connected accelerator.

1. Introduction

Computational efficiency is one of the primary goals in any processor design. Architects try to design processors that execute applications as quickly as possible, while using as few resources as possible (e.g., die area or energy consumed). One common method for achieving computational efficiency in processors is to add specialized hardware accelerators that are specifically designed for a small number of applications. These application-specific integrated circuits, or ASICs, often provide an order of magnitude improvement in terms of performance and power efficiency.

The main drawback with using ASICs is that they have very poor post-programmability. That is, a single ASIC is excellent at improving the efficiency of one application, but not effective across a range of applications. Because of this drawback, many researchers have investigated accelerator designs that are more generalized. Some examples of these programmable computation accelerators include 3-1 ALUs [13, 20], ALU pipelines [5], closed-loop ALUs [22], and *function* units [24].

Each of these accelerators has shown that they are effective at increasing the efficiency of computation across a range of applications. However, these accelerator designs only target computation that is easily supported in hardware. This leaves much room for improvement when looking at the problem from the applications' perspective. Work by Clark et al. [8] took a slightly different approach, designing an accelerator targeting important computation patterns, whether or not they were easily supported in hardware. This accelerator, termed a configurable compute accelerator (CCA), offers the promise of more efficiency if the hardware can be constructed.

The CCA is essentially a two dimensional array of function units, where input data comes into the top row, is processed by each row, and exits through the bottom row. Each element of the array contains combinational circuitry to perform common integer computations on the data, such as ADD and XOR. Between rows of the array, there is a rich interconnect, enabling any element from the previous row to send data to any element of the subsequent row. As mentioned previously, this design was shown to effectively accelerate the important computation across a wide range of applications.

The problem with this design is that the hardware is difficult to build. The large number of function units and the interconnect consume a great deal of die area. In some applications, the CCA primarily operates on 8-bit data types, making the majority of the accelerator datapath unutilized. Experiments have also shown that the rich interconnect is underutilized in most applications. Beyond the die area overhead, the wide function units and rich interconnect can also have a negative effect on processor clock cycle.

This paper attempts to reconcile these problems. We present a new accelerator design, based on the CCA. This new accelerator has narrow function units and a reduced interconnect in order to support as many computations as possible, while making it more feasible to build in hardware. The narrow function units are coupled with logic that dynamically detects when input data is too large to execute (e.g., 32-bit data being processed on an 8-bit accelerator). This logic

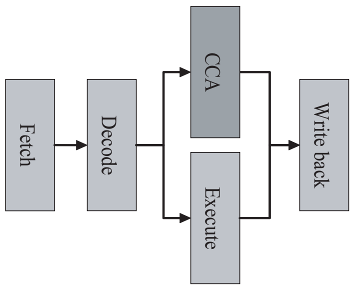

Figure 1. Sample pipeline with CCA.

emulates a wider accelerator by computing the wide results through iteratively generating narrow results (e.g., computing a 32-bit ADD by generating four 8-bit ADDs). This design provides a way to accelerate many common computations, without all of the hardware implications of previously proposed designs.

This new design has many implications for compiling applications targeting the accelerator. Generally speaking, the main compilation challenge in generating code for accelerators is determining which portions of an application to execute on the accelerator and which portions to leave on the standard pipeline. Some researchers have looked into this problem before, proposing greedy algorithms [5, 14], exact methods with exponential runtimes [16, 17], or exact methods in conjunction with heuristics to avoid degenerate cases [10]. Here, previously proposed compiler algorithms are extended to take into account the reduced interconnect and the data-centric latency of the proposed accelerator design. Reduced interconnect makes it more difficult for the compiler to determine which portions of the application can execute on the accelerator. Narrow width function units mean that the accelerator latency is a function of the size of input data, complicating the compile-time decision of whether a computation is more efficient to execute on the accelerator or on the baseline pipeline. These problems must be taken into account to ensure high quality utilization of the accelerator.

This paper demonstrates that narrowing function units and reducing the interconnect makes accelerators more feasible to build in hardware. It also shows how the compiler must be modified to ensure high quality code generation when targeting such accelerators. To summarize, this paper makes the following contributions:

- It presents the design of a novel computation accelerator. This accelerator can execute the most common computations across a wide range of applications, but uses a fraction of the hardware cost of previously proposed accelerators.
- It develops new compiler techniques to deal with the proposed accelerator. This includes modifying the compiler to statistically handle data-dependent computation latency, and take into account reduced interconnect.
- It evaluates the proposed accelerator with respect to the previously proposed CCA design, and demonstrates the effectiveness of the compiler algorithms.

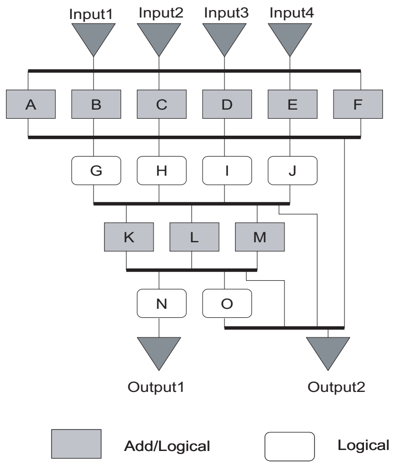

Figure 2. General CCA configuration.

2. Design of a Reduced CCA

In many processor designs, delay and area of certain accelerators make them infeasible to use. The proposed narrow accelerator is a suitable candidate whenever delay or area of the wide accelerators is beyond the tolerable limit of a processor. The new accelerator is also ideal for applications that do not operate on 32-bit inputs, such as video compression applications. CCA, as proposed in [8], is used as a baseline accelerator in this work. In essence, CCA is combinational acyclic accelerator consisting of a set of function units organized as a matrix.

This section begins with an overview of the pipeline organization necessary to support a CCA. Then, a narrow CCA and CCA with sparse interconnect are presented. Finally, the delay and die area effects of reducing CCA resources are explored.

2.1. Pipeline Organization

Architects have proposed many different styles of accelerator integration to merge accelerators into digital systems. The focus of this work is accelerators tightly coupled with the main pipeline of processor. This type of accelerator usually has direct access to register file and behaves as an extra function unit (FU) in the pipeline. In other words, this type of accelerator can be treated as an addition to the execution stage in the main pipeline, but with more computational power.

Figures 1 and 2 show a CCA, proposed in [8], and how it is integrated into the pipeline from a high level. This extra FU can execute dataflow subgraphs of instructions in one cycle. A CCA-cognizant compiler identifies suitable parts of an application to be executed on the accelerator and marks them in the application binary. Whenever the processor shown in Figure 1 fetches and decodes the first instructions of a marked subgraph, it will send all of the instructions in that subgraph to the CCA. In other words, fetching a new subgraph, hardware creates a complex operation for that subgraph. The created complex instruction would be executed

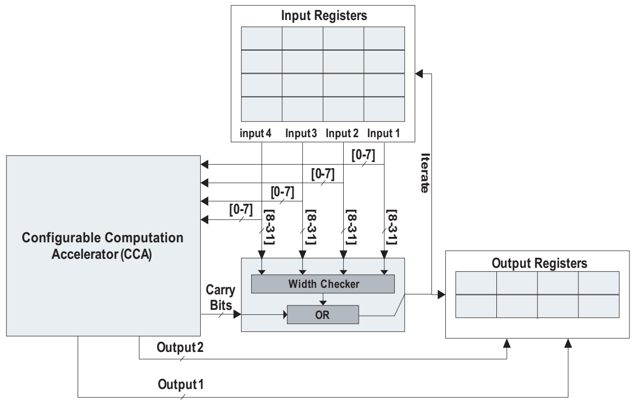

Figure 3. Organization of a width-aware narrow CCA. An 8-bit design is shown.

on the CCA in one cycle. Details about how to fetch a subgraph and create a complex instruction are further discussed in [9].

The CCA consists of number of FUs placed in a matrix. Each element in the row can perform addition/subtraction and/or bit-wise logical operations. The rows get their inputs from the previous rows and can send their results to the next row. Interconnect between rows determines the FU connectivity. The CCA input and output connectivity also depends on the interconnect. A sample CCA is shown in Figure 2. It has 4 inputs, 2 outputs, 4 rows, and 15 FUs. The interconnect between rows is full cross bar, which means each FU in row *i* can send its result to any FU in row *i+1*.

2.2. Width-Aware Narrow CCA

As previously mentioned, this general CCA design has a long critical path and significant die-area issues, making it infeasible in many processor designs. The obvious solution to the critical path problem is pipelining the CCA. One problem with pipelining is that it would make latency for all subgraphs longer, significantly impacting performance gains from the CCA. Previous work showed that moving from the original CCA to 2-stage pipelined CCA eliminates around 30% of the performance improvement [8]. Another problem is that pipelining does not reduce the die area of CCA; it would actually make the problem worse due to pipeline registers.

A different strategy for tackling the CCA's area and delay issues is to reduce the datapath width of FUs. That is, only provide a small 8 or 16-bit datapath instead of the full 32 or 64-bit datapath. This dramatically reduces the area requirements of the CCA, and also improves latency. The downside of this change is that subgraphs that require the full datapath cannot natively be run on the accelerator, reducing potential performance improvements.

However, this performance penalty can be somewhat mitigated by emulating subgraphs that require a wider datapath than what is supported by the CCA. To clarify, if a subgraph is 8-bit, it can be executed in one cycle on an 8-bit CCA. If a subgraph requires 16 or 32-bits of datapath, then it can be executed in 2 or 4 cycles respectively by computing each 8-bit chunk of the result separately. The only additional hardware needed to support wide subgraph emulation is a width checker for operands and a carry register for each FU.

Comparing a width-aware CCA to a pipelined CCA, the narrow, width-aware CCA has the advantage that its latency is only as slow as the data requires it to be, where pipelining uniformly slows down all subgraphs. A pipelined CCA is also much larger, in terms of die area, than the reduced width CCA. Cost savings and its data-centric latency makes the narrow, width-aware CCA a suitable choice for situations where the original CCA is too large or too slow to implement.

As previously mentioned, the additional hardware required to emulate wider computation on a narrow datapath is modest. First, each FU in the CCA needs to store extra data from instructions that are not bit-wise. For example, add/subtract instructions require at most one carry bit from low order bits to emulate a wider datapath. Shift-left emulation would require significantly more registers. These carry bits keep the state of computation after each iteration. In addition to the carry bits, a width checker is needed in order to determine the width of inputs. This dynamic width checker is used in conjunction with the carry bits to determine if more iterations of the datapath are needed to compute a final result. If, for example, one of the inputs is 16-bit and the rest are 8-bit, then an 8-bit CCA might need two or three iterations to compute the results. Two iterations will be sufficient if there is no carry-out after the second iteration. If there is a carry out in any of FU nodes, an extra iteration is needed to get the final outputs. This extra hardware enables the width-aware CCA to emulate all of the wider operations that the general CCA can perform with the exception of shift-right.

The design for a width-aware, 8-bit CCA is shown in Figure 3. Once data enters the accelerator, the width checker determines the width of inputs. After that, the result of an OR operation between width checker output and carry bits determines if more iterations are needed. In each iteration, input and output registers get shifted to right. This ensures the first 8 bits of the input registers always contain the data for next iteration. If the last 24 bits of all inputs are zero (for positive numbers) or one (for negative numbers), and there are no carry bits set, then execution of subgraph is completed. Two registers are used at the output in order to store and shift the partial results. CCA subsystem indicates the end of computation by setting the iterate signal to zero. At this point, results are sent to the main pipeline for writeback. Synthesis results in Section 2.4 show that the overhead of the width checker and carry bits is small.

In order to better illustrate how a narrow CCA works, an example using a 4-bit CCA is illustrated in Figure 4. The target CCA, shown in Figure 4a, has three 4-bit FUs. Each of these FUs can perform basic ALU operations. The small circle beside each unit of this CCA is the carry register for that unit. These carry register hold the state of computation between iterations. In this example, we want to perform $[(0x1D + 0x0C) + (0x20\ OR\ 0x08)]$ on this CCA.

Figure 4b shows the CCA after assignment of operations to units. The 4-bit CCA requires at least two iterations to execute this computation, because maximum input width is 8-bits. Figure 4c shows the first iteration. The first four bits

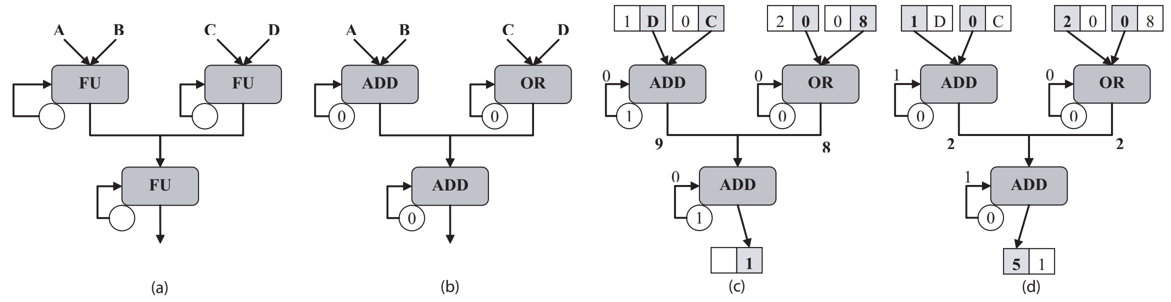

Figure 4. (a) A sample width-aware 4-bit CCA with 4 inputs and 1 output. (b) The initial configuration of the CCA before starting computation. (c) First iteration of an 8-bit computation. (d) Second iteration of an 8-bit computation.

of all input numbers are used as input to the FUs. This iteration computes the first four bits of the output and the carry bits needed for the next iteration. The second iteration operates on the most significant four bits of the inputs. Figure 4d shows the state and the outputs generated after the second iteration. At this point, because all of the bits of the input numbers are consumed and there are no carry bits set in FUs, the CCA subsystem writes results back to the main pipeline.

Using a narrow datapath CCA provides significant die area savings over the original CCA design, however, it also has a significant impact on the compiler. The narrow datapath means that the size of the inputs will affect the execution latency. This makes determining what portions of an application to execute on the CCA much more difficult. Section 3 will discuss how to effectively account for this data-centric latency.

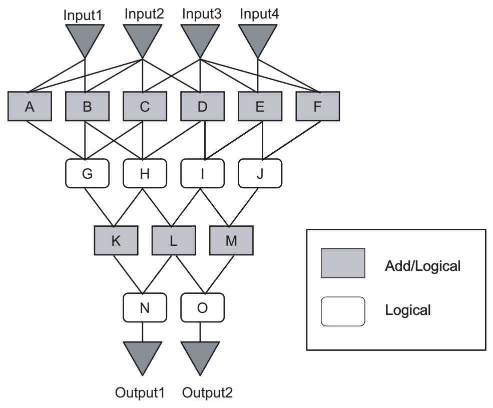

Figure 5. CCA with sparse interconnect.

2.3. Sparse Interconnect CCA

Besides datapath width, another way to reduce the area and delay in CCAs is by making the interconnect more sparse. The original CCA design had a full crossbar interconnect between rows. First of all, this interconnect impacts the delay and increases the overall area. Second, our experiments show that most of the subgraphs can be executed on far sparser interconnects.

Figure 5 shows a CCA with sparse interconnect. The only difference in this CCA, compared to the original one in Figure 2, is the interconnect. In order to find a good design point for sparse CCA, we measured the utilization of each point-to-point connection in each row. Connections that are underutilized (i.e., used less than 2% of the time) have been removed from the interconnect network. The sparse interconnect CCA in Figure 3 is able to run more than 91% percent of important subgraphs identified in targeted applications, which made it a good design point from our experiments.

Sparse interconnect does not affect how subgraphs execute, but it does make the compilation problem more difficult. The sparse interconnect makes it harder to determine whether or not the CCA can support a given computation. In the CCA with full interconnect, it is fairly easy to find an assignment of nodes to execute a computation (or to prove that no such assignment exists), because each node within a row is identical. Sparse interconnect changes that, though. Again, Section 3 will discuss what the compiler must do to handle this new situation.

2.4. Hardware Synthesis Results

In order to clearly understand effects of narrowing the datapath and making the interconnect more sparse, we synthesized different accelerator configurations using a Synopsys synthesis tool chain and a 90nm standard cell library. Table 1 shows these results. The trend that jumps out of the table is that narrowing the datapath has a very significant effect on both latency and area of the CCA. The latency of 8-bit CCA with full interconnect is 30% less than the latency of 32-bit CCA with full interconnect. The 8-bit CCA was also nearly four times smaller than 32-bit CCA.

Pruning the CCA interconnect did not have quite as drastic effect on the latency and area, but still offered solid improvement over the CCA with a full crossbar. Table 1 also shows that width checker has a very small area and delay comparing to the CCA itself. This is important so that the benefits of reduced datapath width are not overshadowed by the costs of the checker. Overall, this table shows that reducing datapath width and pruning interconnect meets the goal of making CCAs easier to integrate into a high-performance pipeline.

Accelerator Configuration	Latency(ns)	Area(mm^2)
32-bit CCA with Full Interconnect	3.30	0.301
32-bit CCA with Sparse Interconnect	2.95	0.270
16-bit CCA with Full Interconnect	2.88	0.168
16-bit CCA with Sparse Interconnect	2.55	0.140
8-bit CCA with Full Interconnect	2.56	0.080
8-bit CCA with Sparse Interconnect	2.00	0.070
Width Checker	0.39	0.002

Table 1. Delay and area of different accelerator configurations.

3. Compilation for CCA

Compiling an application to make use of computation accelerators boils down to two steps: *enumerating* portions of the application's dataflow graph (DFG) that can be executed on the accelerator, and *selecting* which portions to accelerate.

Enumeration consists of generating a set of subgraphs from a given DFG, and determining if they can run on an accelerator. Generating a set of subgraphs is difficult because the number of possible subgraphs grows exponentially with the size of the DFG. Determining if the subgraphs can run on an accelerator, i.e., determining if they perform the same computation, is essentially equivalence checking, which is NP-complete [12]. The problem is further complicated if the accelerators perform a superset of the desired computation (e.g., an accelerator for dot-products could also accelerate multiply-accumulates in an application).

Selecting which subgraphs to accelerate is also difficult. Typically, the selection problem is formulated to push as much computation as possible onto the accelerators, while minimizing overlap between subgraphs. That is, given a set of enumerated subgraphs, find the group that covers the largest portion of the DFG while minimizing the number of nodes appearing in multiple subgraphs. This problem is also NP-complete and is quite similar to the well known technology mapping problem in VLSI design. Clearly, mapping applications onto accelerators is a challenging compilation problem. Reducing the interconnect and narrowing the FUs only complicates things.

The new compilation algorithms presented here are an extension of previous work on this problem described in [10]. The algorithm from prior work was termed FEU, for Full Enumeration - Unate covering. The remainder of this section will briefly describe this algorithm, and then discuss extensions needed to support effective compilation for the reduced accelerators proposed in Section 2.

3.1. Full Enumeration - Unate Covering Subgraph Mapping

As mentioned at the beginning of this section, mapping portions of an application onto accelerators is a difficult problem. Because of this, greedy subgraph mapping algorithms are the most widely used strategy in compilers today. Greedy algorithms are attractive because of their fast runtimes. However, they are also suboptimal in many cases [16]. The FEU algorithm uses more thorough solution-space exploration techniques, while employing intelligent heuristics to avoid exponential runtime that can occur in any NP-complete problem.

The FEU algorithm performs subgraph mapping one basic (or super) block at a time, and consists of four phases: subgraph enumeration, pruning, unate covering, and grouping.

The first step, enumeration, generates the set of all connected subgraphs within a block that can potentially be executed by the target accelerator. At first glance, this seems an intractable problem: in the most general sense, each operation in a DFG could either be included or excluded in a potential subgraph candidate, yielding 2^N potential subgraphs. However, very fast techniques for enumerating connected subgraphs for acceleration have been developed in the past few years, e.g., [3, 4]. The FEU algorithm takes advantage of these techniques to quickly generate candidate subgraphs. These candidates could potentially not be executable on the accelerator; enumeration simply generates candidates subject to the input/output constraints of the accelerator.

After full enumeration, FEU prunes away invalid candidate subgraphs using more detailed checks that are hard to incorporate into enumeration. The purpose of pruning is to ensure that candidates can actually be executed on the accelerator. This takes into account functionality and connectivity issues that were ignored during enumeration. The method employed to determine that subgraphs can execute on an accelerator is based on subgraph isomorphism. Loosely stated, subgraph isomorphism determines whether or not a subset of the nodes in a particular graph are equivalent to a separate graph. In this case, a graph representing the hardware structure is constructed, and we attempt to find a subset of hardware vertices that can create a computation equivalent to the subgraph discovered in enumeration. If we find such a subset, then the dataflow subgraph is capable of being executed on the accelerator.

Once a set of subgraphs that *can* execute on the accelerator is developed, unate covering selects which subgraphs *to* execute on the accelerator. Informally speaking, unate covering problems operate on a Boolean matrix, M, where the rows represent vertices in a DFG, and the columns represent subgraphs; if the value of $M_{i,j}$ is True, this means that operation i occurs in subgraph j. The goal of unate covering is to find a set of columns (or subgraphs) with minimal cost, such that each operation is covered at least once. In this formulation, the cost of a subgraph could be a variety of things, such as the number of cycles needed to execute on a particular accelerator or the power consumed by a subgraph.

After unate covering selects a set of subgraphs to execute on the accelerator, a grouping phase combines subgraphs that can be executed in parallel. Recall that enumeration only creates connected dataflow subgraphs. Often, if the selected subgraphs are small, multiple disjoint subgraphs can execute on an accelerator simultaneously. FEU iteratively groups selected subgraphs until no more combining is feasible, thereby maximizing the size of accelerated subgraphs. Once grouping completes, subgraphs are marked for execution on the accelerator and compilation continues as normal.

To summarize, the FEU algorithm uses more thorough solution-space exploration to identify portions of an application suitable for acceleration than traditional greedy meth-

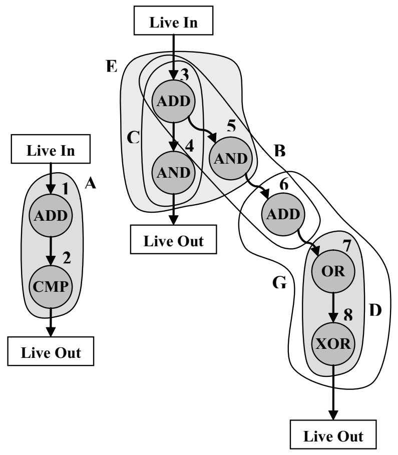

Figure 6. Sample DFG with potential subgraphs marked following full enumeration.

Width	Op ID	A	B	C	D	E	G	H	I	...	O
24	1	1						1		...	
8	2	1							1	...	
24	3		1	1		1				...	
8	4			1		1				...	
32	5		1			1				...	
32	6		1				1			...	
8	7				1		1			...	
8	8				1		1			...	1
Cost		3	4	3	1	4	4	1	1	...	-
Benefit		-1	-1	-1	1	-1	-1	0	0	...	0

Table 2. Unate covering matrix. Rows represent operations, columns represent subgraphs. The first column shows the width of each operation, and the bottom two rows show the cost/benefit for each subgraph using Equation 1.

ods. FEU was also shown to have runtimes that were reasonable, thus providing a good basis for our modifications.

3.2. FEU Example

In order to clarify the FEU algorithm, we will walk through a simplified example using the basic block shown in Figure 6. The first step in the algorithm is to enumerate all connected subgraphs in the DFG, subject to input/output constraints. This will create candidate subgraphs, such as instructions 1 and 2, 3-4, 3-5-6, 6-7-8, etc., until all connected subgraphs are enumerated. Using I/O constraints and removing disconnected subgraphs is essential for keeping the enumeration process tractable.

Once the connected subgraphs are generated, subgraph isomorphism is used to remove candidates that are not supported by the targeted accelerator. This happens by first constructing a graph that represents the computation supported by the targeted accelerator. Isomorphism then tries to find a subset of nodes in this graph that can execute the computation requested by the subgraph. This takes into account both the functionality offered by the hardware substrate, as well as the connectivity between FUs in the accelerator. If the candidate subgraph does map onto a subset of nodes in the accelerator's representative graph, then the accelerator *can* execute the desired computation. For the illustrative purposes, assume that the DFG in Figure 6 represents the algorithm's progress up to this point: the circled regions are the only subgraphs that were enumerated and supported by the targeted accelerator.

These subgraphs are placed into a unate covering matrix (shown in Table 2) to select which subgraphs to execute on the accelerator. In this table, columns represent candidate subgraphs, rows represent operations, and a 1 in element$_{i,j}$ signifies that operation$_i$ is part of subgraph$_j$ (for the moment, disregard the width column and cost rows on the left and bottom, respectively). Unate covering selects a set of subgraphs such that each operation is covered by at least one subgraph and the total number of subgraphs is minimized. In this formulation, each operation is individually added to the subgraph list to ensure a solution can be found (subgraphs H - O in the table). The solution to the unate covering problem in Table 2 will be subgraphs A, E, and G, which totals 3 subgraphs. In the initial CCA work, the number of subgraphs was equivalent to the number of cycles needed to perform the targeted computation.

After unate covering, grouping will attempt to combine A, E, and G to form larger subgraphs. This reduces the number of subgraphs needed to cover the operations, and ultimately improves performance. Assume in this example that the only combinable subgraphs are A and C. If this was the case, then grouping would be a null step, since none of the candidates chosen by unate covering are groupable. Performing the grouping after unate-covering essentially precludes certain disconnected subgraphs from being selected, and is a weakness of the FEU algorithm. However, this did not adversely affect results when targeting the original CCA design.

Once grouping finishes, FEU has generated a set of subgraphs for execution on the targeted accelerator. The compiler marks the selected subgraphs for execution on the accelerator, and the application is effectively retargeted. The remainder of this section describes how this algorithm must be modified to handle the reduced CCA described in Section 2.

3.3. Extensions for Sparse Interconnect

The original proposed CCA, shown in Figure 1a, had a very rich interconnect. We demonstrated in Section 2 that by pruning this interconnect to only include the paths most commonly used, it is possible to reduce the area and critical path of the CCA.

The problem with reducing the interconnect is that it puts more onus on the pruning used in FEU to ensure that subgraphs can be executable. Recall that subgraph isomorphism is used to perform subgraph pruning. That is, a graph representing the accelerator is constructed, and the pruner tries to identify a set of nodes in the hardware graph that are capable of executing the candidate subgraph.

In the CCA from Figure 1a, with full interconnect and identical functionality within a row, the subgraph isomorphism problem degenerates to only having to identify which row each operation in the candidate subgraph is placed in. To rephrase, since all hardware graph nodes within a row have identical functionality and identical interconnect, it does not matter which node an operation from the candidate subgraph is assigned, only which row. This makes pruning invalid candidates much easier, as it greatly limits the number of potential solutions isomorphism must examine.

As an example, assume the compiler wanted to determine if the subgraph for $a = (x + y) \oplus (z\&w)$ was executable on a CCA. With the CCA from Figure 1a, isomorphism would identify that the ADD and AND can be supported in row 1 of the CCA, the XOR is supported by row 2 of the CCA, and therefore this subgraph can be executed on the CCA. However, if the compiler is targeting the CCA in Figure 5, it needs to identify that the ADD is supported on node A, the AND on node B, the XOR on node G, and that there is sufficient interconnect between all these nodes.

Essentially, this means that the FEU algorithm must model much more of the hardware during pruning, and cannot leverage the special-case characteristics that made the fully-connected unate covering problem easier to solve. Using established algorithms for subgraph isomorphism [15, 23], we found that this more complicated pruning did not significantly affect algorithm runtime.

Note that this more complicated pruning technique is an enabling technology, meaning that it must be done to correctly compile targeting the proposed family of accelerators. It does not affect performance of the resulting code.

3.4. Extensions for Width-Sensitive Latency

Like pruned interconnect, reducing the width of the datapath in a CCA reduces the area and cycle time. The downside is that subgraphs that use the entire datapath now have a longer latency. From the compiler's perspective, this makes selecting subgraphs to execute on the accelerator more difficult, because the latency depends on the dynamic width of the data being processed. There are two places in FEU that this impacts: the cost function used in unate covering, and the phase ordering of the FEU algorithm.

First we will detail the cost function. Unate covering, as described above, tries to minimize the number of subgraphs needed to cover every operation in a block. The cost function is simply 1 for each subgraph. This is equivalent to speedup in a single-issue in-order processor, since each covered node is replaced by an accelerator instruction that takes one cycle. However, this latency assumption is no longer true with data-dependent accelerator latency. Many of the subgraphs that would be selected with the old cost function might not be beneficial if they are small and frequently use the full 32-bit datapath.

The first step in designing a new algorithm, termed the data-centric algorithm, is designing an appropriate cost function that takes into account data width. The factors that affect benefit of a subgraph for a width-aware CCA is the number of instructions in the subgraph and maximum width of instruction inputs and outputs. Maximum width determines the number of iterations needed to perform computation in a subgraph. In order to get the width for all of the computation instructions in a program, we performed width profiling. For each operation, we recorded the percentage of time its inputs and outputs widths were less than or equal 8 bits, larger than 8 bits but less than or equal 16 bits, larger than 16 bits but less than or equal 24 bits, and finally larger than 24 bits (assuming the target processor is a 32-bit processor and that the accelerator has an 8-bit datapath). These data give an estimate for how many iterations will likely be needed for each operation on a width-aware CCA. The cost function uses the profile information and size of subgraph to estimate benefit of each subgraph, and it can be formulated as:

$$\begin{aligned}
W_k^{i,j} &= Percentage\ of\ time\ that \\
&\quad\ i\ < (Width\ inst\ k)\ \leq\ j, \\
A^k &= (W_k^{0,8} \times 1 + W_k^{9,16} \times 2 + W_k^{17,24} \times 3 + \\
&\quad\ W_k^{25,32} \times 4), \qquad (1) \\
Cost(S) &= max(A^k)\ \forall\, instruction\ k \in subgraph\ S, \\
Benefit(S) &= Number\ of\ instructions(S) - Cost(S)
\end{aligned}$$

The $Cost$ function in Equation 1, for subgraph S computes the expected number of cycles for the width-aware CCA to execute the subgraph. If the $Benefit$ for a particular subgraph is greater than zero, this subgraph is better executed on the width-aware CCA. For example, for a subgraph consists of three 8-bit instructions, cost function and benefit function would return 1 and 2, respectively. This means executing the subgraph on width-aware 8-bit CCA would take 1 cycle and save 2 cycles compared to executing the three instructions on the main pipeline. Benefit is used to remove subgraphs from the unate covering formulation that would not lead to effective solutions in narrow CCAs.

Upon initial experimentation, we were surprised how effective using the average width of an operation was as a means of predicting the latency of subgraphs. If the width of an operation had changed a lot during execution, then this estimate could potentially hurt compilation results. Consider the case of two operations each with an average datapath usage of 16 bits. If the first operation always used 16 bits, but the second operation used 28 bits half the time and 4 bits the other half of the time, then creating a subgraph with these two operations would unnecessarily penalize the operation that always used 16 bits, since occasionally it would be run for four iterations.

Figure 7 shows why using the proposed cost function is sufficient, though. The x-axis of this figure lists a number of benchmarks that were width profiled using a training input. The y-axis lists the variance of the data size on a per operation basis in all of the enumerated subgraphs. A value of 1 for a benchmark means that the data width of every operation in the application never changed throughout execution of the training run. To rephrase, this graph shows that static operations that require only an 8-bit datapath in one dynamic execution, rarely require more than 8 bits of datapath in a different execution. Likewise, operations that require 24 bits of datapath, will probably require 24 bits of datapath for each dynamic execution.

Note that every benchmark had variance above 85%. The

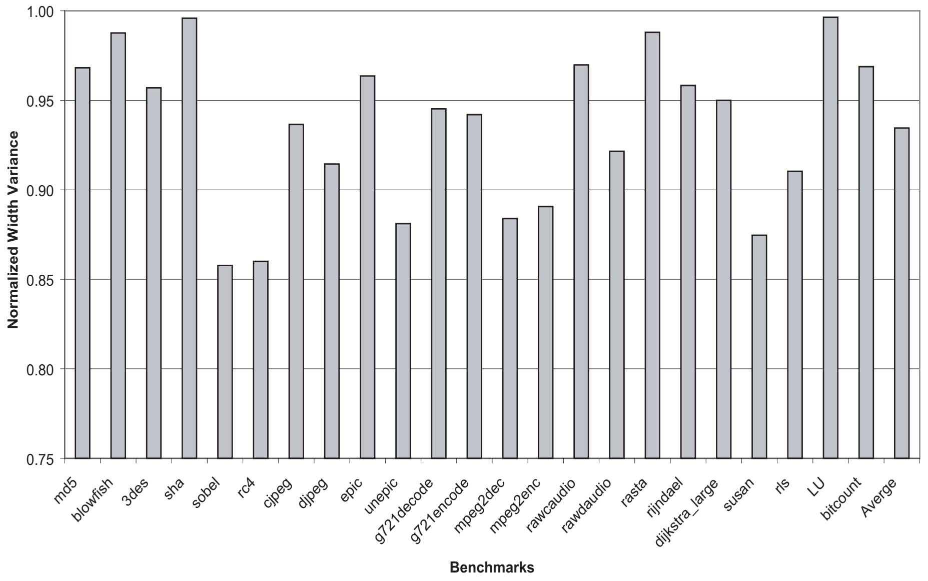

Figure 7. Variance of the data width of operations for a number of applications. A value of 1 signifies that the data width remained in the same range (0 to 8, 9 to 16, 17 to 24 or 25 to 32), on a per operation basis, throughout execution of the program.

majority of the benchmarks were well above 90%. The implication of this is that A^k, the average latency of an instruction on a narrow CCA, is an effective measure of the dynamic width. Because of this, more complicated heuristics tracking data width correlation between operations are unnecessary for the benchmarks examined.

Now that a new cost function is defined, we reexamine the phase ordering of the FEU algorithm. Recall from the FEU example above that grouping performed after selection can result in finding suboptimal solutions for width-aware narrow CCA. We found that, unlike the original CCA, grouping after selection significantly affects the quality of solutions when targeting reduced datapath CCAs.

To overcome this shortcoming, we propose generating all possible disconnected subgraphs, through grouping, prior to unate covering selection. This exposes all potential candidate subgraphs to the selection engine ensuring that the best possible solution is discovered. The downside of this is that it can greatly increase the size of the unate covering matrix, potentially impacting compile time. Experiments show that the impact is not significant, though.

3.5. Data-centric Algorithm Example

To illustrate the benefits of the proposed algorithm changes, this section revisits the example from Figure 6. In the unate covering table for this example (Table 2), the bottom rows contain the cost and benefit of the executing of each subgraph using Equation 1. Again, this cost represents the number of execution cycles a subgraph would need to execute on CCA. Subgraphs H - O contain only a single operation, and thus their cost is 1, since they will never be run on the accelerator (these exist only to ensure a solution in the unate covering). All of the subgraphs with negative benefits are removed from the table before the unate covering, but after grouping because they will not definitely be part of an optimal solution.

It is clear that the new cost function is necessary to prevent subgraphs from hurting performance. For example, using the old cost function, subgraphs C and D (in Table 2) could both potentially be selected, since they reduce the number of subgraphs selected in the application. However, taking width into account, neither of them would be selected, as they hurt performance in the average case.

To demonstrate the importance of grouping before selection, consider the FEU solution in Table 2. Without knowledge of subgraphs that can be grouped, unate covering selected subgraphs A, E, and G as the final solution. This precluded the disconnected subgraph A+C from being discovered (assuming A and C can be grouped together), and resulted in an expected execution time of 11 cycles in this block.

Now consider the unate covering matrix shown in Table 3, which contains all disconnected pre-grouped subgraphs as well. After grouping, all subgraphs that are expected to hurt performance are removed from the table. Each individual instruction is still part of the table with cost 1 to ensure that a feasible solution exists. The unate coverer does a better job minimizing cost, because A+C is visible to it, where it had no idea this was possible in the previously proposed algorithm. Ultimately, the modified FEU algorithm will discover the better solution of subgraphs A+C, D, L and M, yielding an execution time of 6 cycles per block instead of 11 cycles. This example demonstrates the benefits of using width-cognizant cost function, and the importance of grouping before selection.

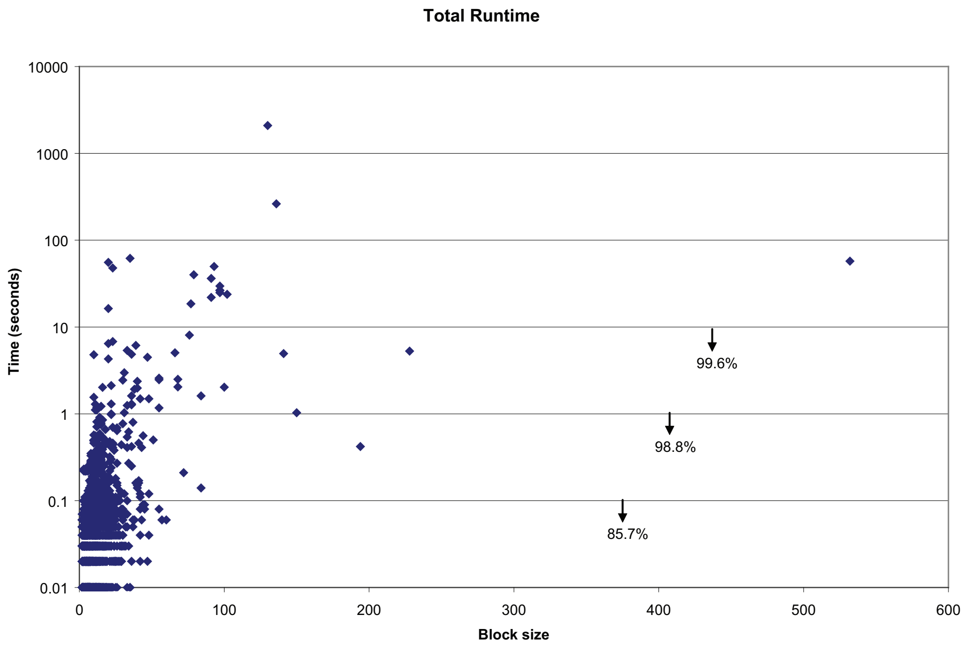

Figure 8. Runtime of the FEU algorithm with proposed extensions. Each dot represents one block from a targeted application.

Width	Op ID	D	A+C	H	I	...	O
24	1		1	1		...	
8	2		1		1	...	
24	3		1			...	
8	4		1			...	
32	5					...	
32	6					...	
8	7	1				...	
8	8	1				...	1
Cost		1	3	1	1	...	1
Benefit		1	1	0	0	...	0

Table 3. Unate covering matrix with grouping prior to selection. Dominated columns were removed for clarity.

3.6. Algorithm Runtime

There are clearly benefits in compilation quality by using the FEU algorithmic improvements that we have proposed for narrow-datapath accelerators. The major concern is that the proposed improvements do not make compilation times intractable. This is an issue because pruning has become more complex, and grouping prior to unate covering makes the unate covering matrix much larger. Figure 8 demonstrates that compilation times are indeed reasonable.

Each point in this graphs represents the algorithm runtime of a basic block from MediaBench and MiBench applications. The data was collected on a Core 2 DUO machine with 2 GB of RAM. Applications were compiled to target an 8-bit accelerator with 4 inputs, 2 outputs, and a maximum dependence height of 4 (shown in Figure 5). Overall, more than 85.7% of blocks took less than 0.1 second to compile. 98.8% of basic blocks took less than 1 seconds total.

The worst case block (by an order-of-magnitude) out of the 23 applications took around 30 minutes for all phases. This was primarily because of a degenerate case in the unate covering phase. Several techniques can be used in order to reduce this time, such as giving the algorithm an artificial timeout or partitioning the problem into covering multiple smaller portions of that particular block. Both of these techniques could hinder achieved speedups, but ultimately provide tractable runtimes without affecting the vast majority of applications. We leave exploration of these techniques for future work. In conclusion, Figure 8 demonstrates that, in the common case, our algorithm finds the optimal solution in less than 1 second for a majority of applications.

4. Experiments

In order to evaluate the proposed hardware and compiler algorithms, an experimental framework was built using the Trimaran research compiler and SimpleScalar ARM simulator. Trimaran was re-targeted for the ARM instruction set and subgraphs to be accelerated were marked in the binary. After compilation, the simulator recognized the subgraphs and modeled them as if an accelerator was present. SimpleScalar was configured to represent an ARM-926EJ-S [2], a popular embedded core.

Several benchmarks from the MediaBench and MiBench suites are used to demonstrate different aspects of the proposed algorithm and hardware. Omitted benchmarks are due to issues in compiler infrastructure and simulator, not lim-

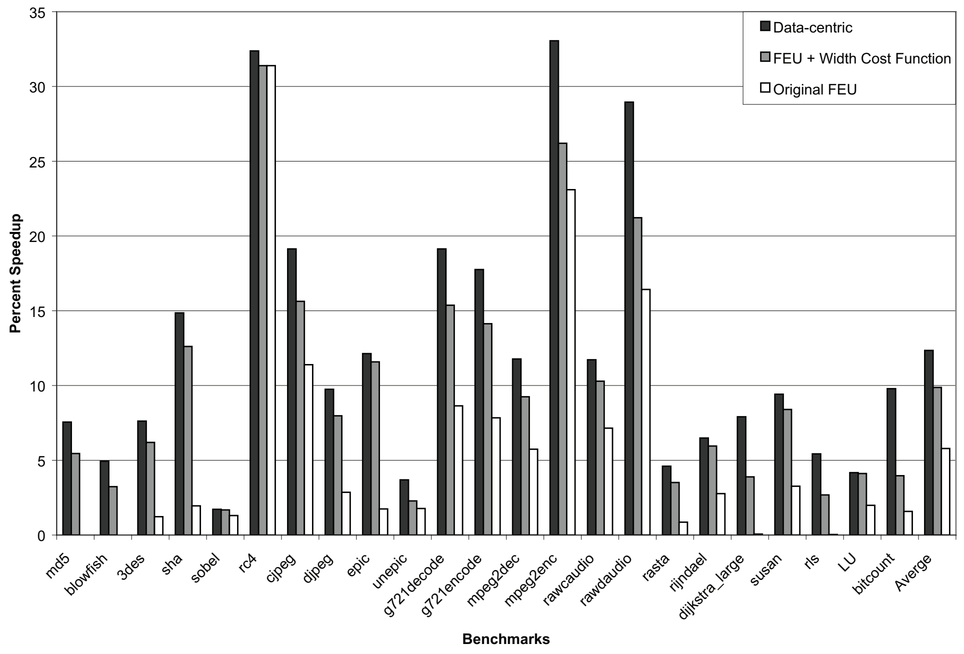

Figure 9. Comparison of the speedup for original FEU algorithm, FEU algorithm with the width-aware cost function and data-centric algorithm containing both width-aware cost and pre-selection grouping. Compilation targeted a width-aware 8-bit CCA.

itations of the algorithms. Three different experiments are performed in this section. First, the speedups due to use of FEU algorithm and data-centric algorithm are compared to illustrate the effectiveness of the new data-centric algorithm. Second, the speedup for different accelerator widths is studied. Finally, the data-centric algorithm is used to juxtapose the performance results between full-interconnect and sparse-interconnect CCAs.

Compilation Algorithm Comparison: In order to illustrate the effectiveness of the data-centric algorithm for width-aware narrow CCAs, all of the benchmarks were compiled with the original FEU algorithm, the FEU algorithm with our new cost function, and the data-centric algorithm. The results of this experiment are shown in Figure 9.

Overall, the data-centric algorithm achieves more than 6.5% performance improvement over the original FEU algorithm. The data-centric algorithm also improves performance more than 2.5% over the FEU algorithm with the new cost function. Encryption benchmarks, such as SHA, 3DES, and MD5, show large performance improvement when the compilation algorithm is changed from the original FEU to the data-centric. This trend is expected, because encryption application primarily operate on effectively random data, making the width of subgraphs very large. In this type of application, selection and grouping without considering the width of subgraphs, leads to very small speedup or even performance loss. In RC4, all of the algorithms nearly perform the same because execution time of RC4 is mostly spent in a small for-loop that does calculations on 8-bit data. Therefore, it is not difficult for the FEU algorithm with or without width-aware cost function to select beneficial subgraphs. Rawdaudio has a mix of 8-bit and 32-bit computation in its main loop. The FEU algorithm with the width-aware cost function causes 4% less performance improvement than the data-centric algorithm. This difference is expected in any benchmark with mixed narrow and wide computations.

Effectiveness of Data-Centric Algorithm for Different CCA widths: Now that the effectiveness of the new compiler algorithm has been established, we can evaluate the performance for the proposed CCA architecture changes. In the first experiment, we compare speedup of a 32-bit 1-cycle CCA, a width-aware 16-bit and a width-aware 8-bit CCA using the data-centric compilation technique. The results are shown in Figure 10. Note that this graph compares the number of clock cycles each benchmark takes and not the total execution time (i.e., the data does not account for the clock speed).

As would be expected 32-bit CCA has the highest performance and width-aware 8-bit CCA has the lowest performance. In a number of the benchmarks, such as RC4 and Rawcaudio, the total execution cycles is the same for width-aware 16-bit and 32-bit CCA. The same trend can be seen in dijkstra_large when an 8-bit CCA is used instead of a 16-bit CCA. On average, reducing the CCA datapath from 32 to 16 and 8 bits increases the total number of cycles by 10% and 21% comparing to the 32-bit CCA. If we assume the CCA constitutes the processor's critical path, however, then the 16-bit and 8-bit CCA will perform 7% and 9% better, respectively, than the 32-bit version, because of the lesser impact on clock cycle. Therefore, it can be concluded that, using the data-centric algorithm, the 8-bit CCA is the most benficial accelerator among all three CCAs.

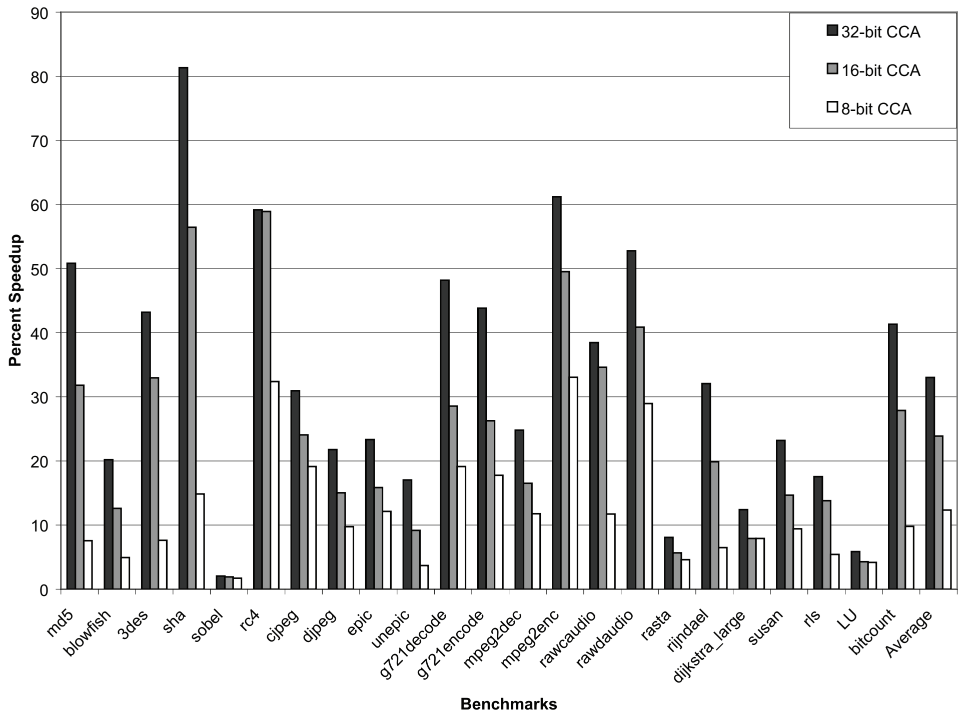

Figure 10. Speedup using the data-centric algorithm for various width CCAs. Each CCA had 1-cycle assumed latency, regardless of clock cycle reported in Table 1.

CCA Interconnect Comparison: As discussed in Section 2, making CCA interconnection sparse can be an effective way to reduce CCA area and delay. The downside is that it could also preclude important subgraphs from being executed and hurt performance. Figure 11 shows the effect of changing full-interconnect to sparse-interconnect on the 32-bit 1 cycle CCA.

Overall, we found that changing the interconnect from full crossbar to relatively sparse caused nearly no performance loss (if we assume equal clock speeds) in any of the benchmarks. The reason for this negligible performance loss is that the sparse-interconnect CCA in Figure 5 can support most of the important subgraphs that original CCA supports. In cases where an important subgraph cannot be executed because of sparse interconnect, enumeration ensures that slightly modified subgraphs are selected, picking up any available slack. If we again assume that the CCA is the processor's critical path, then the sparse CCA actually performs 14% better than the CCA with full crossbar. This experiment demonstrates that with the aid of a compiler, the cost and delay of the CCA can definitely be reduced without affecting the performance.

5. Related Work

Compiling an application to make use of computation accelerators boils down to two steps: *enumerating* portions of the application's dataflow graph (DFG) that can be executed on the accelerator, and *selecting* which portions to accelerate.

The vast majority of previous work relies on hand coding or greedy heuristics. Work by Hu [14] is typical of the greedy solutions: a seed node is selected in the DFG and is grown along dataflow edges. The compiler then replaces that subgraph and repeats the process. Here, enumeration consists of finding a seed and growing it, while selection is implicit (any subgraph enumerated is automatically selected). Other previous work [22] performs more thorough enumeration, but still uses greedy selection.

More thorough, traditional code generation methods for tackling subgraph mapping use a tree covering approach [1]. In this approach, all computation subgraphs potentially supported by the accelerator must be constructed a priori. During compilation, the DFG is split into several trees. The trees are then covered by the computation subgraphs using an algorithm that minimizes the number of computation subgraphs used. The purpose behind splitting the DFG into trees first is that there are linear time algorithms to optimally cover trees, making the process very quick.

The major problem with this method is that many DFGs and accelerators are not trees. It is shown in [16] that tree covering methods can yield suboptimal results, particularly in the presence of irregular computation commonly targeted by embedded systems. To overcome this, [16] proposes splitting all instructions into "register-transfer" primitives and recombining the primitives in an optimal manner using integer programming. Work by Liao [17] attacked the same problem and developed an optimal solution for DFG covering by augmenting a binate covering formulation. While both of these solutions are optimal, they also have worst case exponential runtime and do not report how long their algorithms take.

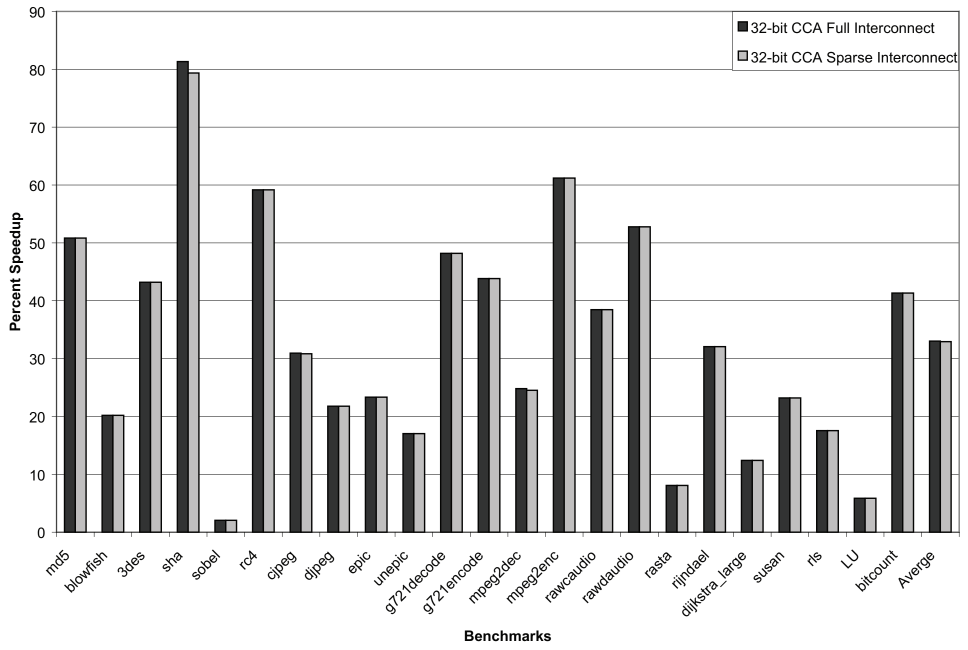

Figure 11. Comparison of 32-bit full interconnect and sparse interconnect CCAs using the data-centric algorithm.

Another major problem with previously mentioned approaches is that they also require permissible accelerator subgraphs to be enumerated a priori. If an accelerator supports a wide range of computations, such as an ALU pipeline, this can cause an explosion in runtime.

Research in [19] describes a different way to look at the accelerator mapping problem. In this work, an application is initially decomposed into an algebraic polynomial expression that is functionally equivalent to the original application. Next, the polynomial is manipulated symbolically in an attempt to use accelerators as best as possible. For example, a polynomial could be expanded using function identities (e.g., adding 0 to a value) to better fit an accelerator. This enables the algorithm to utilize subgraphs where the accelerator performs a superset of the desired computation. As with previous solutions, though, this technique also has exponential worst-case runtime. Additionally, handling bit-wise operations, such as XOR, is difficult using polynomials. Rearranging application to better fit a targeted accelerator, such as [19] proposes, is an interesting area of future work, though.

The main differentiator between this paper and prior work is that no previous work on accelerator mapping has taken into account dynamic data width before. We also extend previous work to explore a larger solution space (through preselection grouping) without adversely affecting compilation time.

Datapath width on the architectural side has been extensively studied for different reasons, such as increasing performance or reducing energy consumption. Work in [7] proposes operand gating for improving processor energy efficiency by gating off the sections of data path that are unneeded by narrow operands. In this work, the instruction set architecture is enhanced to include opcodes that specify operand widths. A compiler or binary translator uses statically available information to determine value ranges and generate efficient code. Research in [21] evaluates managing the processor's datapath width at the compiler level by means of exploiting dynamic narrow-width operands.

Several dynamic approaches have also been proposed to make use of narrow data computations in different application domains. Authors in [11] propose register packing to reduce register file pressure. This approach packs multiple results that have fewer significant bits than full width of a register into a single register. Work by Loh [18] investigates datawidth locality. In this research, a Multi-Bit-Width (MBW) microarchitecture is proposed that that can increase the effective issue width of a superscalar by simultaneously executing several narrow-width computations. Very low power pipelines using significance compression is proposed in [6]. In this work, data, address, and instructions are compressed by maintaining only significant bytes with two or three extension bits appended to indicate significance byte positions. Their approach to significance compression is similar to our approach for detecting input widths.

This work harnesses previous work on narrow datapaths and dynamic width detection to make the CCA feasible to build in hardware.

6. Conclusion

Computation accelerators provide an effective way to improve the performance and efficiency of processor designs. However, many styles of accelerator are expensive in terms of area and delay. This paper presents two effective techniques to make accelerators smaller and faster: narrowing the function unit datapath and reducing the interconnect. We demonstrate that accelerators with narrow datapaths and sparse interconnect can be constructed 64% faster, using only 20% of the die area of a full accelerator.

These reduced accelerators require new compilation techniques to utilize them, though. To accomplish this, we propose three extensions to a previously proposed algorithm: modeling interconnect using subgraph isomorphism, an improved cost function that considers operand size, and a phase-ordering change that exposes more disconnected subgraphs to the selection algorithm. Experiments show that the new data-centric compilation algorithm performs on average 6.5% better and up to 12% better than previous algorithms for width-aware 8-bit accelerators. Using this new algorithm, accelerators with narrow datapaths and sparse interconnect perform better than full datapaths and interconnect, but without the associated hardware costs.

7 Acknowledgments

Many thanks to Shantanu Gupta, who provided us with assistance in synthesizing the various accelerators. Additional thanks go to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Limited, the National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] A. Aho, M. Ganapathi, and S. Tijang. Code generation using tree pattern matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[2] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, Jan. 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of the 40th Design Automation Conference*, pages 256–261, June 2003.

[4] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 1246–1251, 2005.

[5] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 18–29, Dec. 2004.

[6] R. Canal, A. Gonzalez, and J. E. Smith. Low power pipelines using significance compression. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 181–190, 2000.

[7] R. Canal, A. Gonzalez, and J. E. Smith. Software-controlled operand-gating. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, page 125, Washington, DC, USA, 2004. IEEE Computer Society.

[8] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[9] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[10] N. Clark et al. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, page To Appear, Oct. 2006.

[11] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 304–315, Washington, DC, USA, 2004. IEEE Computer Society.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[13] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar cisc processors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 213–226, 2006.

[14] S. Hu and J. E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 213–226, 2004.

[15] E. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.

[16] R. Leupers and P. Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proc. of the 1996 European Design Automation Conference*, pages 200–205, Sept. 1996.

[17] S. Liao et al. Instruction selection using binate covering for code size optimization. In *Proc. of the 1995 International Conference on Computer Aided Design*, pages 393–399, 1995.

[18] G. H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 395–405, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[19] A. Peymandoust et al. Automatic instruction set extension and utilization for embedded processors. In *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, pages 108–120, June 2003.

[20] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3):257–268, 1994.

[21] G. Pokam, O. Rochecouste, A. Seznec, and F. Bodin. Speculative software management of datapath-width for energy optimization. In *Proc. of the 2004 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[22] P. Sassone, D. S. Wills, and G. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proc. of the 2005 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, June 2005.

[23] J. R. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[24] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.