# Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization

Nathan Clark[1], Manjunath Kudlur[1], Hyunchul Park[1], Scott Mahlke[1], and Krisztián Flautner[2]

[1]Advanced Computer Architecture Lab
University of Michigan - Ann Arbor
[2]ARM Ltd.
Cambridge, United Kingdom

## ABSTRACT

Application-specific instruction set extensions are an effective way of improving the performance of processors. Critical computation subgraphs can be accelerated by collapsing them into new instructions that are executed on specialized function units. Collapsing the subgraphs simultaneously reduces the length of computation as well as the number of intermediate results stored in the register file. The main problem with this approach is that a new processor must be generated for each application domain. While new instructions can be designed automatically, there is a substantial amount of engineering cost incurred to verify and to implement the final custom processor. In this work, we propose a strategy to transparent customization of the core computation capabilities of the processor without changing its instruction set. A configurable array of function units is added to the baseline processor that enables the acceleration of a wide range of dataflow subgraphs. To exploit the array, the microarchitecture performs subgraph identification at run-time, replacing them with new microcode instructions to configure and utilize the array. We compare the effectiveness of replacing subgraphs in the fill unit of a trace cache versus using a translation table during decode, and evaluate the tradeoffs between static and dynamic identification of subgraphs for instruction set customization.

## 1. INTRODUCTION

In embedded computing, a common method for providing performance improvement is to create customized hardware solutions for particular applications. For example, embedded systems often have one or more application-specific integrated circuits (ASICs) to perform computationally demanding tasks. ASICs are very effective at improving performance, typically yielding several orders of magnitude speedup along with reduced energy consumption. Unfortunately, there are also negative aspects to using ASICs. The primary problem is that ASICs only provide a hardwired solution, meaning that only a few applications will be able to fully benefit from their functionality. If an application changes, because of a bug fix or a change in standards, the application will usually no longer be able to take advantage of the ASIC. Another drawback is that even when an application can utilize an ASIC, it must be specifically rewritten to do so. Rewriting an application can be a large engineering burden.

Instruction set customization is another method for providing enhanced performance in processors. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to specialized hardware. Though not as effective as ASICs, instruction set extensions improve performance and reduce energy consumption of processors. Instruction set extensions also maintain a degree of system programmability, which enables them to be utilized with more flexibility. An additional benefit is that automation techniques, such as the ones used by ARM OptimoDE, Tensilica, and ARC, have been developed to allow the use of instruction set extensions without undue burden on hardware and software designers.

The main problem with application specific instruction set extensions is that there are significant non-recurring engineering costs associated with implementing them. The addition of instruction set extensions to a baseline processor brings along with it many of the issues associated with designing a brand new processor in the first place. For example, a new set of masks must be created to fabricate the chip, the chip must be reverified (using both functional and timing verification), and the new instructions must fit into a previously established pipeline timing model. Furthermore, extensions designed for one domain are often not useful in another, due to the diversity of computation causing the extensions to have only limited applicability.

To overcome these problems, we focus on a strategy to customize the computation capabilities of a processor within the context of a general-purpose instruction set, referred to as *transparent instruction set customization*. The goal is to extract many of the benefits of traditional instruction set customization without having to break open the processor design each time. This is achieved by adding a configurable compute accelerator (CCA) to the baseline processor design that provides the functionality of a wide range of application-specific instruction set extensions in a single hardware unit. The CCA consists of an array of function units that can efficiently implement many common dataflow subgraphs. Subgraphs are identified to be offloaded to the CCA and then replaced with microarchitectural instructions that configure and utilize the array.

Several different strategies are proposed for accomplishing transparent instruction set customization. One strategy, a fully dynamic scheme, performs subgraph identification and instruction replacement in hardware. This technique is effective for preexisting program binaries. To reduce hardware complexity, a static strategy performs subgraph identification offline during the compilation process. Subgraphs that are to be mapped onto the CCA are marked in the program binary to facilitate

simple CCA configuration and replacement at run-time by the hardware.

The contributions of this paper are twofold:

- We present the design of the CCA, which provides the functionality of common application specific instruction set extensions in a single hardware unit. A detailed analysis of the CCA design shows that it implements the most common subgraphs while keeping control cost, delay, and area overhead to a minimum.
- We describe the hardware and software algorithms necessary to facilitate dynamic customization of a microarchitectural instruction stream. The tradeoffs of these algorithms are discussed and the effectiveness of each is experimentally determined.

## 2. RELATED WORK

Utilizing instruction set extensions to improve the computational efficiency of applications is a well studied field. Domain specific instruction set extensions have been used in industry for many years, for example Intel's SSE or AMD's 3DNow! multimedia instructions. Techniques for generating domain specific extensions are typically adhoc, where an architect examines a family of target applications and determines what is appropriate.

In contrast to domain specific extensions, many techniques for generating application specific instruction set extensions have been proposed [1, 5, 8, 14, 16, 28]. Each of these algorithms provide either exact formulations or heuristics to effectively identify those portions of an application's DFG that can efficiently be implemented in hardware. While these techniques provide a good baseline for our work, they are not directly applicable. One reason is that these techniques are computationally intensive, meaning that performing them dynamically on chip would have a substantial performance overhead. Additionally, these techniques select computationally critical subgraphs without any assumptions regarding the underlying computational hardware. This is a valid assumption when designing new hardware; however, these techniques need to be modified to map onto a CCA.

A great deal of work has also been done on the design of a reconfigurable computation accelerators. Examples include PRISM [2], PRISC [25], OneChip [6], DISC [29], GARP [15], and Chimaera [30]. All of these designs are based on a tightly integrated FPGA, which allows for very flexible computations. However, there are several drawbacks to using FPGAs. One problem is that the flexibility of FPGAs comes at the cost of long latency. While some work [20] has addressed the issue, implementing functions in FPGAs remains inefficient when compared to ASICs that perform the same function. Second, FPGA reconfiguration time can be slow and the amount of memory to store the control bits can be large. To overcome the computational inefficiency and configuration latency, the focus of most prior work dealing with configurable computation units was on very large subgraphs, which allows the amortization of these costs. This work differs in that we focus on acceleration at a finer granularity.

Recent research [31] has proposed using a finer granularity CCA based on slightly specialized FPGA-like elements. By restricting the interconnect of the FPGA-like elements, they reduce the delay of a CCA without radically affecting the number of subgraphs that can be mapped onto the accelerator. While the flexibility to map many subgraphs onto configurable hardware is appealing, there are still the drawbacks of a large number of control bits and the substantial delay of FPGA-like elements.

A key observation we have made is that when collapsing dataflow subgraphs for customized instruction set extensions, the flexibility of an FPGA is generally more than is necessary. FPGAs are designed to handle random computation. The computation in applications is structured using a relatively small number of computational primitives (e.g. add, subtract, shift). Thus, the types of computation performed by instruction set extensions can be implemented much more efficiently by designing a dedicated circuit corresponding to primitives from dataflow graphs. Constructing a circuit of dataflow graph primitives has the additional benefit of keeping the configuration overhead to a bare minimum. This is because selecting from a few primitives is far simpler than selecting from all possible computations. By sacrificing some generality, we are able to achieve a much simpler architecture that still captures the majority of subgraphs.

REMARC [21] and MorphoSys [19] are two designs that also proposed computation architectures more suited for computation of DFG primitives than an FPGA. These coprocessors were geared toward large blocks in multimedia applications, as compared to our design, which executes smaller blocks of computation. Both REMARC and MorphoSys must be programmed by hand to be effectively utilized, since they target large blocks of very regular computation.

Work proposed in [24] presents the design of a special ALU, which has the capability to execute multiple dependent DFG primitives in a single cycle. This idea is more in line with our proposal than most papers on configurable compute architectures, but our work takes an entirely different perspective. Their work only seeks to reduce the dependence height in DFGs. This is demonstrated in their design decision to only allow configurable computations that have restricted shapes and at most three inputs and one output. Recent work [32] has shown that limiting configurable computations in this manner on many potential performance improvements in several domains that were not explored in [24]. Here, we present the design of a CCA which takes advantage of more general computation subgraphs.

Once a CCA has been designed, it becomes necessary to map portions of an application onto the accelerator. Two examples of using the compiler to statically map dataflow subgraphs onto a CCA are [8] and [23]. Both of these techniques target fixed hardware, and it is not clear if the algorithms extend to cover CCAs not exposed to the instruction set.

Several frameworks have been proposed which would lend themselves to dynamically mapping dataflow subgraphs onto configurable accelerators. Dynamo [4], Daisy [11], and Transmeta's Code Morphing Software [10] are all schemes that optimize and/or translate binaries to better suit the underlying hardware. These systems can potentially do a much better job of mapping an application to CCAs than compile time systems, since they can take advantage of runtime information, such as trace formation. Using these systems has the additional benefits that algorithms proposed to statically map computation to a CCA would be effective, and full binary compatibility is provided.

| | Encryption | | | MediaBench | | | | SPECInt | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Depth | crc | blowfish | rijndael | djpeg | g721enc | gsmenc | unepic | gzip | vpr | parser | vortex | Average |
| 2 | 11.13 | 10.37 | 4.17 | 29.79 | 42.51 | 41.57 | 74.87 | 39.19 | 44.37 | 50.39 | 38.07 | 47.53 |
| 3 | 11.27 | 72.29 | 77.75 | 38.91 | 69.38 | 41.57 | 95.23 | 53.48 | 46.07 | 82.20 | 63.49 | 72.30 |
| 4 | 22.37 | 81.42 | 77.75 | 100.00 | 69.38 | 41.57 | 100.00 | 62.21 | 95.49 | 82.54 | 100.00 | 82.61 |
| 5 | 22.37 | 99.98 | 100.00 | 100.00 | 84.71 | 45.84 | 100.00 | 73.40 | 99.99 | 82.54 | 100.00 | 88.85 |
| 6 | 100.00 | 100.00 | 100.00 | 100.00 | 84.71 | 48.77 | 100.00 | 95.46 | 100.00 | 100.00 | 100.00 | 95.53 |
| 7 | 100.00 | 100.00 | 100.00 | 100.00 | 87.24 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 99.47 |
| ≥8 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |

**Table 1: Cumulative percentage of dynamic subgraphs with varying depths**

Many hardware based frameworks exist for mapping application subgraphs onto CCAs, as well. Most of these frameworks arose from the observation that, in systems with a trace cache, the latency of the fill unit has a negligible performance impact until it becomes very large[13] (on the order of 10,000 cycles). That is, once instructions retire from the pipeline and traces are constructed, there is ample time before the traces will be needed again. Both Instruction Path Coprocessors [7] and rePLay [12] propose taking advantage of this latency to optimize the instruction stream.

DISE [9] is another framework that could potentially support dynamic mapping of application subgraphs onto a CCA. In DISE, Application Customization Functions specify how to translate the instruction stream of an application. One of the ways DISE differs from the Instruction Path Coprocessor and rePLay, is that DISE is performed in the decode stage of the pipeline. This implies that mapping algorithms would have to be simpler in a DISE implementation than in a fill unit based design, since decode is more timing critical.

While all of these frameworks provide an effective place to map application subgraphs onto CCAs, none have proposed methods to do so. Two works do discuss dynamically mapping subgraphs onto CCAs [17, 26]. These papers use the design proposed in [24] as the baseline of their system, which greatly simplifies the problem. Because they used a more restrictive CCA design, the identification algorithm did not need to be as complex as the one we propose. An additional difference between our work and [17, 26], is that we provide a method for the mapping engine to replace subgraphs that were statically determined, as well as dynamically determined. This functionality is extremely useful, since offline mapping algorithms can be more thorough than their online counterparts.

# 3. DESIGN OF A CONFIGURABLE COMPUTE ACCELERATOR

The main goal of a CCA is to execute many varied dataflow subgraphs as quickly as possible. A matrix of function units (FUs) is a natural way of arranging a CCA, since it allows for both the exploitation of parallelism in the subgraph and also for the sequential propagation of data between FUs. In order to be effective, the FUs need to have adequate functionality to support the types of operations that are frequently mapped onto them.

A set of experiments were performed to determine the depth (number of rows) and the width (number of columns) of the matrix of FUs, as well as the capabilities of each FU. Using the SimpleScalar toolset [3] for the ARM instruction set, traces were collected for a set of 29 applications. The application set consisted of four encryption related algorithms and selected MediaBench and SPECint benchmarks. The goal of this benchmark set was to represent a wide variety of integer computational behavior.

Traces from these benchmarks were analyzed offline using the optimal discovery algorithm (described in section 4.2) to determine the important subgraphs a CCA should support. The characteristics of these subgraphs were then used in determining the configuration of our proposed CCA. The subgraphs were weighted based on execution frequency to ensure that heavily utilized subgraphs influenced the statistics more. Because dynamic traces are used as the basis for analysis, conservative estimates have to be made with regards to which operation results must be written to the register file. That is, unless a register is overwritten within the trace, it must be written to the register file, because it may be used elsewhere in the program. This potentially restricts the size of subgraphs available to offline replacement schemes, however it accurately reflects what is necessary for supporting runtime replacement techniques.

The subgraphs considered in this study were limited to have at most four inputs and two outputs. Further, memory, branch, and complex arithmetic operations were excluded from the subgraphs as will be discussed later in the section. Previous work [32] has shown that allowing more than four input or two output operands results in very modest performance gains when memory operations are not allowed in subgraphs, thus the input/output restriction is considered reasonable.

A similar characterization of subgraphs within traces was performed previously [27]. This differs from the analysis here in that we gear experiments specifically toward the design of a CCA. The previous work proposed many additional uses for frequently occurring subgraphs, for example cache compression and more efficient instruction dispersal.

## 3.1 Analysis of Applications

The matrix of FUs comprising a CCA can be characterized by the depth, width, and operation capabilities. Depth is the maximum length dependence chain that a CCA will support. This corresponds to the potential vertical compression of a dataflow subgraph. Width is the number of FUs that are allowed to go in parallel. This represents the maximum instruction-level parallelism (ILP) available to a subgraph. The operation capabilities are simply which operations are permitted in each cell of the matrix.

**Depth of Subgraphs:** Table 1 shows the percentage of subgraphs with varying depths across a representative subset of the three groups of benchmarks. For example, the 81.42% in blowfish at depth 4 means that 81.42% of dynamic subgraphs in blowfish had depth less than or equal to 4. Although only 11 benchmarks are displayed in this table, the final column displays the average of all 29 applications run through the system. On average about 99.47% of the dynamic subgraphs have depth 7 or less.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 100.00 | 59.02 | 22.89 | 13.14 | 6.48 | 4.20 | 0.25 |
| 2 | 91.11 | 50.57 | 9.93 | 4.10 | 0.59 | 0.15 | 0.01 |
| 3 | 57.39 | 17.79 | 6.25 | 2.89 | 0.09 | 0.02 | 0.01 |
| 4 | 18.53 | 8.27 | 1.58 | 0.11 | 0.02 | 0.01 | 0.00 |
| 5 | 8.65 | 2.06 | 0.14 | 0.04 | 0.01 | 0.01 | 0.00 |
| 6 | 2.13 | 1.23 | 0.09 | 0.01 | 0.01 | 0.00 | 0.00 |
| 7 | 1.23 | 0.10 | 0.07 | 0.01 | 0.00 | 0.00 | 0.00 |
| 8 | 0.11 | 0.07 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 |

**Table 2: Matrix utilization of subgraphs**

| Uop | Opcode Semantics | Percentage |
|---|---|---|
| ADD | addition | 28.69 |
| AND | logical AND | 12.51 |
| CMP | comparison | 0.38 |
| LSL | logical left shift | 9.81 |
| LSR | logical right shift | 2.37 |
| MOV | move | 11.66 |
| OR | logical OR | 8.66 |
| SEXT | sign extension | 10.38 |
| SUB | subtract | 4.82 |
| XOR | logical exclusive OR | 5.09 |

**Table 3: Mix of operations in common subgraphs**

Since the depth of the CCA directly affects the latency through it, depth becomes a critical design parameter. It can be seen that a CCA with depth 4 can be used to implement more than 82% of the subgraphs in this diverse group of applications. Going below depth of 4 seriously affects the coverage of subgraphs implementable by the CCA. Therefore, only CCAs with maximum depth of 4 to 7 are considered.

**Width of Subgraphs:** Table 2 shows the average width statistics of the subgraphs for the 29 applications. A value in the table indicates the percentage of dynamic subgraphs that had an operation in that cell of the matrix layout (higher utilized cells have a darker background). For example, 4.2% of dynamic subgraphs had width 6 or more in row 1. Only 0.25% of subgraphs had width 7 of more, though. Similar cutoffs can be seen in the other rows of the matrix, such as between widths 4 and 5 in row 2. This data suggests that a CCA should be triangular shaped to maximize the number of subgraphs supported while not needlessly wasting resources.

**FU Capabilities:** Table 3 shows the percentage of various operations present in the frequent subgraphs discovered in above set of benchmarks. Operations involving more expensive multiplier/divider circuits were not allowed in subgraphs, because of latency considerations. Additionally, memory operations were also disallowed. Load operations have non-uniform latencies, due to cache effects, and so supporting them would entail incorporating stall circuitry into the CCA. This would increase the delay of the CCA and make integration into the processor more difficult.

Table 3 shows that 48.3% of operations involve only wires (e.g. SEXT and MOV) or a single level of logic (e.g. AND and OR). Another 33.9% of operations (ADD, CMP, and SUB) can be handled by an adder/subtracter. Thus, the adder and the wire/logic units were the main categories of FUs considered for the design of a CCA. Although shifts did constitute a significant portion of the operation mix, barrel shifters were too large and incurred too much delay for a viable CCA.

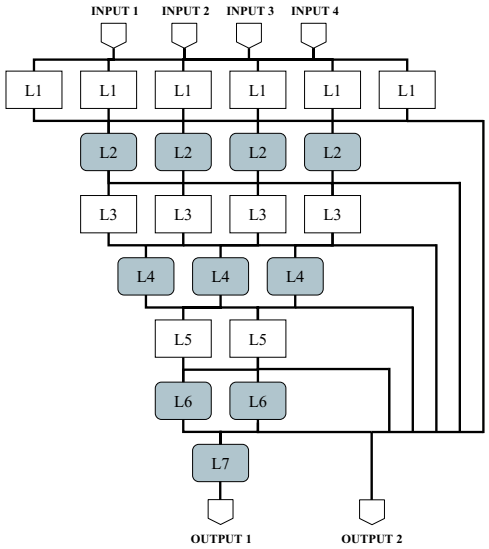



**Figure 1: Block diagram of the depth 7 CCA**

## 3.2 Proposed CCA Design

The proposed CCA is implemented as a matrix of heterogeneous FUs. There are two types of FUs in this design, referred to as type A and B for simplicity. Type A FUs perform 32-bit addition/subtraction as well as logical operations. Type B FUs perform only the logical operations, which include and/or/xor/not, sign extension, bit extraction, and moves. To ease the mapping of subgraphs onto the CCA, each row is composed of either type A FUs or type B FUs.

Figure 1 shows the block diagram of a CCA with depth 7. In this figure, type A FUs are represented with white squares and type B FUs with gray squares. The CCA has 4 inputs and 2 outputs. Any of 4 inputs can drive the FUs in the first level. The first output delivers the result from the bottom FU in the CCA, and the second output is optionally driven from an intermediate result from one of the other FUs.

The outputs of the FUs are fully connected to the inputs of the FUs in the subsequent row. The decision to only allow units to talk to the next row was made to keep the amount of control to a minimum. As the outputs of one row and the inputs of the next are fully connected, the interconnect network is expensive in terms of delay. This delay was necessary, however, to reduce the complexity of the dynamic discovery and selection algorithms described in the next section.

The critical path of adder/subtracter circuits is much longer than any of the other operations supported by the CCA. To control the overall delay, the number of rows with adders is restricted. More than 99.7% of dynamic subgraphs can be executed on a CCA with 3 adders in serial, and so the depth 7 CCA in Figure 1 is restricted to having 3 rows of type A FUs. Further, restricting the CCA to only 2 rows of type A FUs allows it to support only 91.3% of the subgraphs, but significantly improving the delay of the CCA. The type A and type B rows were interspersed within the CCA, because empirical analysis shows many of the subgraphs perform a few logic operations between subsequent additions. This is particularly true in the encryption applications.

Four CCA models were synthesized using Synopsys CAD tools with a popular standard cell library in 0.13$\mu$ tech-

| Depth | Configuration | Control | Delay | Cell area | FPGA delay |
|---|---|---|---|---|---|
| 4 | 6A-4B-3A-2B | 172 bits | 3.19 ns | 0.38 $mm^2$ | 18.84 ns |
| 5 | 6A-4B-4A-2B-1B | 197 bits | 3.50 ns | 0.40 $mm^2$ | 19.97 ns |
| 6 | 6A-4B-4A-3B-2A-1B | 229 bits | 4.56 ns | 0.45 $mm^2$ | 24.86 ns |
| 7 | 6A-4B-4A-3B-2A-2B-1B | 245 bits | 5.62 ns | 0.48 $mm^2$ | 25.39 ns |

**Table 4: CCA configurations and synthesis results**

nology. Each model has different depth and row configurations, shown in Table 4. The configurations in this table indicate the number and type of FUs in each row, from top to bottom. For example, the depth 4 CCA has 6 type A FUs in row 1 and 4 type B FUs in row 2. Delay of the CCA and the die area are also listed in this table. The depth 4 CCA had a latency of 3.19ns from input to output and occupied $0.38mm^2$ of die area. The last column of Table 4 contains the delay of each CCA design when synthesized on an FPGA[1]. It suggests that FPGAs may not be a suitable device for an efficient implementation of the CCA at this granularity of subgraph, though we did not perform any measurements of direct realization of the applications' subgraphs via the FPGA.

The control bits needed for each model are also shown in Table 4. Each FU has four opcode bits that define its functionality. Since the output of each FU is connected to every input port of the FUs in the next level, signals to control the bus are required. The number of those signals corresponds to twice the number of FUs in the next level, considering there are two input ports for each FU and each output could feed each input. Control bits for which FU provides the second output are also needed. The total number of control bits was a critical factor in the design of these CCAs.

## 3.3 Integrating the CCA into a Processor

In the context of a processor, the CCA is essentially just another FU, making integration into the pipeline fairly straightforward. The only datapath overhead consists of additional steering logic from reservation stations and bypass paths from the CCA outputs. The CCA itself is not pipelined, removing the complexity of having to introduce latches in the matrix of FUs or having to forward intermediate results from internal portions of the matrix.

Accommodating a 4 input, 2 output instruction into the pipeline is slightly more complicated. One potential way to accomplish this is to split every CCA operation into 2 uops, each having 2 inputs and 1 output. By steering the 2 uops consecutively to a single CCA, a 4 input, 2 output instruction can be constructed without altering register renaming, the reservation stations, the re-order buffer, or the register read stage. The downside to this approach is that the scheduling logic is complicated by having to guide the two uops to the same CCA.

Interrupts are another issue that must be considered during CCA integration. The proposed CCA was intentionally designed using simple FUs that cannot cause interrupts. However, splitting the CCA operation into 2 uops means that an external interrupt could cause only half of the operation to be committed. To avoid this problem, the 2 uops must be committed atomically.

Control bits for the CCA can be carried along with the 2 uops. Since there is at most 245 bits of control necessary in the proposed CCAs, this means that each uop would carry around 130 bits, which is roughly the size of a uop in the Intel P6 microarchitecture.

[1]Xilinx Virtex-II Pro family, based on $0.13\mu$ technology

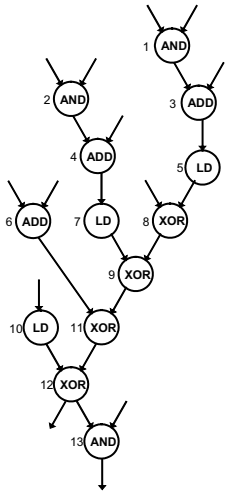
# 4. UTILIZATION OF A CONFIGURABLE COMPUTE ACCELERATOR

Once the CCA is integrated into a processor, it is necessary to provide subgraphs for the CCA to execute. Feeding the CCA involves two steps: *discovery* of which subgraphs will be run on the CCA and *replacement* of the subgraphs with uops in the instruction stream. In this section, two alternative approaches for each of these tasks are presented.

The two proposed approaches for subgraph discovery can be categorized as static and dynamic. Dynamic discovery assumes the use of a trace cache and performs subgraph discovery on the retiring instruction stream that becomes a trace. When the instructions are later fetched from the trace cache, the subgraphs will be delineated. The main advantage of a dynamic discovery technique is that the use of the CCA is completely transparent to the ISA. Static discovery finds subgraphs for the CCA at compile time. These subgraphs are marked in the machine code using two new subgraph specification instructions, so that a replacement mechanism can insert the appropriate CCA uops dynamically. Using these instructions to mark patterns allows for binary forward compatibility, meaning that as long as future generations of CCAs support at least the same functionality of the one compiled for, the subgraphs marked in the binary are still useful. The static discovery technique can be much more complex than the dynamic version, since it is performed offline; thus, it does a better job of finding subgraphs.

The two proposed schemes for replacing subgraphs are both dynamic, but performed at different locations in the pipeline. Replacing subgraphs in the fill unit of a trace cache is the most intuitive place for this task. Previous work [12] has shown that delays in the fill unit of up to 10,000 cycles have a negligible impact on overall system performance. This delay provides ample time for augmenting the instruction stream. The second proposal is to replace subgraphs during decode. The impetus behind this idea was that many microarchitectures (like the Intel Pentium IV) already perform complicated program translations during decode, so subgraph replacement would be a natural extension. The biggest advantage of a decode-based replacement is that it makes the trace cache unnecessary when used in concert with static discovery. Removing the trace cache makes CCAs more attractive for embedded processors, where trace caches are considered too inefficient and power hungry.

The primary reason for using dynamic replacement for CCA instructions is that complete binary compatibility is provided: a processor without a CCA could simply ignore the subgraph specification instructions and execute the instructions directly. This idea extends to future processors as well. As long as any evolution of a CCA provides at least the functionality of the previous generation, the



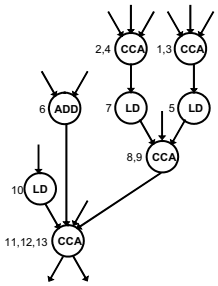

A.

| Operation | Slack |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 0 |
| 6 | 4 |
| 7 | 1 |
| 8 | 0 |
| 9 | 0 |
| 10 | 5 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |

B.

| Step | current_match | priority_queue |
|---|---|---|
| 1 | | 13 |
| 2 | 13 | 12 |
| 3 | 12, 13 | 11, 10 |
| 4 | 11, 12, 13 | 9, 6, 10 |
| 5 | 11, 12, 13 | 6, 10 |
| 6 | 11, 12, 13 | 10 |
| 7 | 11, 12, 13 | |
| 8 | | 9 |
| 9 | 9 | 8, 7 |
| 10 | 8, 9 | 5, 7 |
| 11 | ... | ... |

C.




D.

**Figure 2: A. DFG from a frame in 164.gzip. B. Slack of the operations. C. Trace of Algorithm 1. D. DFG after subgraph replacement.**

statically discovered subgraphs will still be effective. Essentially, this allows for binary compatible customization of the instruction set.

## 4.1 Dynamic Discovery

The purpose of dynamic discovery is to determine which dataflow subgraphs should be executed on the CCA at runtime. To minimize the impact on performance, we propose to use the rePLay framework [22] in order to implement dynamic discovery.

The rePLay framework is similar to a trace cache, in that sequences of retired instructions are stored consecutively and later fetched. RePLay differs because instead of traces, it uses frames, where highly biased branches are converted into control flow assertions. A frame can be thought of as a large basic block, with one entry and one exit point. If any of the control flow assertions are triggered, the entire frame is discarded. This property of rePLay provides an excellent opportunity for subgraph discovery, since subgraphs are allowed to cross control flow boundaries without compensation code. A frame cache also allows for ample time between retirement and when the instruction stream will be needed again.

The algorithm proposed for dynamic subgraph discovery and selection is shown in Algorithm 1. The basic idea underlying this algorithm is to start at an operation not already in a match, and then grow that seed operation toward its immediate parent operations. When parent operations are added to the seed operation, a new subgraph is created for replacement, provided that the subgraph meets the architectural constraints of the CCA. These constraints include number of inputs/outputs, illegal opcodes, and subgraph outputs cannot feed subgraph inputs (necessary to avoid deadlock). An operation's slack (i.e., how critical each operation is to the total dependence height of the DFG) is used to determine the priority of adding operations to the match when multiple parents exist. This heuristic is reminiscent of both Dijkstra's shortest path algorithm or the 'maximal munch' code generation algorithm.

To better illustrate Algorithm 1, Figure 2C shows a sample run on the DFG in Figure 2A. The discovery al-

```
1  for i = N to 1 do
2      if op_i is in a match then
3          Continue;
       end
4      Initialize current_match;
5      priority_queue.push(op_i);
6      while priority_queue not empty do
7          candidate_op ← priority_queue.pop();
8          Add candidate_op to current_match;
9          if current_match does not meet constraints then
10             Remove candidate_op from current_match;
11             Continue;
           end
12         foreach parent of candidate_op do
13             if parent is not in a match then
14                 priority_queue.push(parent);
               end
           end
       end
15     if CCA implementation of current_match is better than
       native implementation then
16         Mark current_match in instruction stream;
       end
17     current_match.clear();
   end
```

**Algorithm 1: Dynamic discovery**

gorithm starts at the bottom operation of the frame with operation 13. Node 13 is popped and added to the match at step 2. Next 13's parent, node 12, is added to the queue and subsequently to the current match. When 12's parents are added to the queue in step 3, note how 11 is ahead of 10 in the queue because it has a slack of 0 as compared to 5. Slacks for all operations are given in Figure 2B. At step 5, node 9 would be added to the match; however, the resulting subgraph would require 5 inputs, which violates the architectural constraints of the CCA. Node 9 is simply discarded and its parents are ignored. This process continues until the priority queue is empty at step 7 and a subgraph is delineated. After the subgraphs are replaced, Figure 2D shows the resulting DFG.

This heuristic guides growth of subgraphs toward the critical path in order to reduce the dependence height of the DFG. The reason subgraphs are only grown toward the parents of operations is because this reduces the complexity of the discovery algorithm, and it guides the shape of the subgraphs to match the triangular shape

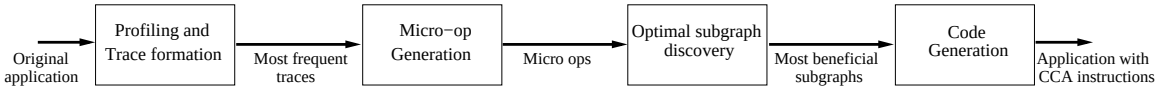



**Figure 3: Workflow of static discovery**

of the proposed CCA design. Note that this algorithm is just a greedy heuristic, and will not perform as well as offline discovery algorithms that have been developed.

## 4.2 Static Discovery

In order to reduce the complexity of the hardware customization engine, a method for offline customization of applications is also proposed. This approach builds on traditional compiler-based techniques for instruction set customization, and is shown in Figure 3. Initially, the application is profiled to identify frequently executed frames. If the execution engine uses microcode, the compiler converts the frames from sequences of architectural instructions to sequences of uops to match what would be seen by the replacement engine. The most frequently executed frames are then analyzed and subgraphs that can be beneficially executed on the CCA are selected. Then, the compiler generates machine code for the application, with the subgraphs explicitly identified to facilitate simple dynamic replacement.

**Trace formation:** A trace is a sequence of basic blocks that are highly likely to be executed sequentially [18]. Traces are identified by profiling the application on a sample input. The trace structure is very similar to the frames that are identified and optimized by rePLay, thus the compiler uses traces as a surrogate for the frames formed by the hardware.

**Micro-operation generation:** In order to identify subgraphs that can be replaced at run-time, the compiler must convert its internal representation to match the runtime instruction stream. For instruction sets such as x86, this implies converting instructions into micro-operations, thereby creating a uop trace. The compiler also keeps track of mapping between instructions and uops to facilitate later code generation. When targeting microarchitectures without uops, this step is unnecessary.

**Optimal subgraph discovery:** The optimal subgraph discovery algorithm used for this paper is based on previous two works [8] and [1]. As described in [8], the subgraph discovery can be logically separated into two phases: (a) candidate enumeration, that is enumerating the candidate subgraphs that can be potentially become a CCA instruction, and (b) candidate selection, that is selecting the beneficial candidates. The branch and bound technique similar to [1] was used to solve the first phase. One additional constraint was added so that all micro-operations for a particular instruction should be included in the subgraph. The selection phase was modeled as an instance of the unate covering problem. All nodes in the DFG corresponding to the trace under consideration have to be covered by the candidate subgraphs so that the overall performance is maximized. The ratio of number of nodes in the original DFG to the number of nodes in the DFG with candidate subgraphs replaced with CCA instructions was used as the performance metric. An additional weight was given to nodes based on their slack so that subgraphs on the critical paths are more likely to be selected.

**Code generation:** After the best subgraphs to execute on the CCA have been identified, the compiler must generate machine code for the application. The objective of this process is to organize the machine code in a manner that facilitates simple dynamic replacement of the uops by CCA operations. To accomplish this two new instructions are introduced into the ISA: $CCA\_START(liveout, height)$ and $CCA\_END$. CCA_START and CCA_END serve as markers for the instructions that comprise a subgraph to be mapped onto the CCA. CCA_START has two operands: liveout is the number of the uop that produces an externally consumed register value, and height is the maximum depth of the micro-operation subgraph. Note that the last uop of the subgraph is assumed liveout, creating a maximum of two outputs. Height is used as a quick feasibility test to efficiently support multiple CCA variations.

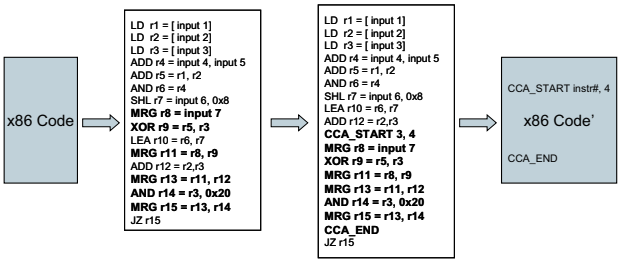



**Figure 4: Static CCA instruction insertion**

For each uop subgraph, the code generator groups the corresponding macro-instructions together. The assembly instructions are topologically sorted based on the structure of the subgraph and placed sequentially in memory. A CCA_START instruction is pre-pended to the list and a CCA_END is post-pended, thereby isolating the subgraph and making it simple for the hardware to discover. For any case where a CCA enabled binary needs to run on a processor without a CCA, the CCA_START and CCA_END instructions are converted to NOPs.

The code generation process is illustrated in Figure 4, which is the x86 instruction/micro-operation view of a DFG from the SPECInt benchmark crafty. The initial trace of x86 code is shown on the left, which is then converted into micro-operations as shown in the second box. A subgraph to be mapped onto the CCA is identified as shown by the darker uops. The code generation process groups the micro operations contiguously, topologically sorts them and inserts the CCA_START and CCA_END operations as shown in the third box. The sequence of micro-operations is then mapped back to augmented x86 instructions that contain the sorted instructions together with the CCA instructions, thereby identifying the micro-operation subgraph at the instruction level.

## 4.3 Subgraph Replacement in Retirement

Replacement is the final step in making use of a CCA, consisting of generating the encoding bits for a given subgraph and substituting them into the instruction stream. As mentioned in Section 3, the encoding of CCA instructions specifies the opcodes for each node of the CCA and the communication between each of the nodes. Determin-

| Pipeline | 4 wide |
|---|---|
| RUU size | 128 |
| Fetch Queue Size | 128 |
| Execution Units | 4 simple ALUs, 2 multipliers, 2 memory ports |
| Branch Predictor | 12-bit gshare |
| Frame Cache | 32k uops, 256 inst traces |
| L1 I-cache | 32k, 2 way, 2 cycle hit |
| L1 D-cache | 32k, 4 way, 2 cycle hit |
| Unified L2 | 1M, 8 way, 12 cycle hit |
| Memory | 100 cycle hit |
| Frame Cache Discovery and Replacement Latency | 5000 cycles |

**Table 5: Processor configuration**

ing the communication of nodes requires one top-down pass over the operations to determine producer/consumer relationships. Placing individual operations at nodes in a CCA can also be done with one pass over the operations by placing each node in the highest row that can support the operation while honoring data dependencies. In the case where back-to-back additions are needed, but not supported by the CCA, move operations are inserted to pass data from the first addition to the second.

As mentioned previously, the rePLay pipeline is an excellent place to perform subgraph replacement for the CCA. Taking advantage of frames allows the replacer to create subgraphs that cross control flow boundaries. Additionally the latency tolerance of a frame cache allows ample time for replacement to take place.

## 4.4 Subgraph Replacement in Decode

The other alternative is to replace subgraphs during decode. This technique has smaller hardware overhead - as the frame cache is unnecessary - but decode-based schemes are more sensitive to latency and do not allow subgraphs to cross basic block boundaries.

One possible solution to the latency issue is to take the burden of generating control bits for CCA instructions out of the decode stage. To accomplish this, we propose allowing a certain number of subgraphs to be predefined in the binary and saved into a translation table when an application loads. The CCA_START instructions could then just store a pointer into this table for the encoding bits, making replacement trivial. The obvious benefit is that this scheme has very low hardware overhead. However, there is an additional constraint that the number of subgraphs that can be used for the entire program is limited by the size of the translation table.

# 5. EXPERIMENTAL EVALUATION

The proposed discovery and replacement schemes were implemented in the SimpleScalar simulator [3] using the ARM instruction set. Within SimpleScalar, some ARM instructions are broken into micro-operations, e.g., load multiple, which performs several loads to a continuous sequence of addresses. Many ARM instructions allow for an optional shift of one operand, and it is important to note that these shifts are also broken into uops. Since our CCA does not support shifts, it would otherwise not be possible to execute these operations on the CCA.

The simulated processor model is a 4-issue superscalar with 32k instruction and data caches. More details of the configuration are shown in Table 5. Consistent with Section 3, the benchmarks used in this study consist of 29 applications from SPECint, MediaBench, and four encryption algorithms. We select a representative subset of the applications to show in our graphs, consisting of four SPECint applications (175.vpr, 181.mcf, 186.crafty, and 255.vortex), six MediaBench applications (djpeg, cjpeg, epic, mpeg2enc, rasta, and rawdaudio) and four popular encryption applications (3des, blowfish, rijndael and rc4). Each benchmark was run for 200 million instructions, or until completion. The initial 50 million instructions of each SPEC benchmark were skipped to allow the initialization phase of the benchmark to complete. All of the benchmarks were compiled using gcc with full optimizations.

Figure 5 compares the performance across varying depth CCAs using the offline discovery algorithm and retirement-based replacement. Speedups are calculated as the ratio of execution cycles without and with the CCA of the specified configuration. The configuration of the CCAs match the descriptions in Table 4 and all have a latency of one. From the graph, the most obvious result is the flatness of each set of bars. Little performance is gained as larger CCA designs are utilized. However, this result could be anticipated as it agrees with the depth statistics observed in Table 1. Generally, adding depth to the subgraph beyond 4 provides only modest gains in coverage (depth 4 covers 82%). Further, the large, important subgraphs that would have been executed on the 7-deep CCA can simply be broken into two subgraphs executed on the smaller CCAs. As long as there is enough ILP and the large subgraph is not on the critical path, this additional reduction of latency achieved with a larger CCA will not significantly improve performance.

There are a number of notable exceptions to the flat behavior. For example, some benchmarks show a performance jump at one particular CCA size. For instance, blowfish from depth 4 to depth 5. This is because a critical subgraph was admitted at that point. Interestingly, sometimes adding depth actually hurts the performance, as in the case of cjpeg. This is because of second order effects involved with subgraph discovery. Sometimes creating a CCA operation out of a large 7-deep subgraph, while optimal from the coverage standpoint, is not as effective as creating two smaller subgraphs.

Figure 6 shows the affect of CCA latency on overall performance. This graph reflects static discovery, retirement-based replacement and a CCA of depth 4. Speedup is calculated in the same manner as in the previous graph. This figure shows that the effect of CCA latency is highly dependent on the application. For example, rc4's speedup rapidly declines when the latency is increased, reaching zero for latency 3 and beyond. This is because rc4 has one dominant critical path on which all the subgraphs appear. Since the subgraphs are all on the critical path, the performance is highly sensitive to the number of cycles to execute each one.

On the other hand, 186.crafty suffers little penalty from the added latency of the CCA. This behavior is generally attributed to one of two reasons. First, the critical path is memory bound, thus CCA latency is a second order effect. Second, the application has enough ILP so that longer CCA latencies are effectively hidden. Such applications benefit from more efficient execution provided by the CCA, but are less sensitive to latency. Other applications, such as 3des and rawdaudio, degrade slightly at small latencies (e.g., 1-3 cycles), then fall off sharply at larger latencies (e.g., 4 or 5 cycles). This reflects the point

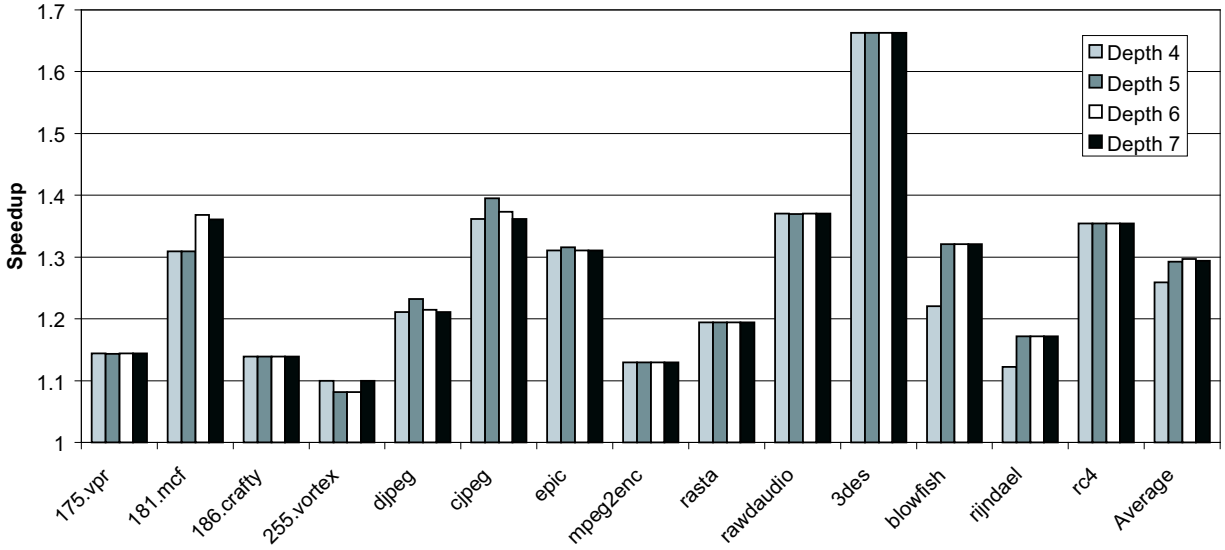



**Figure 5: Varying the CCA configurations**

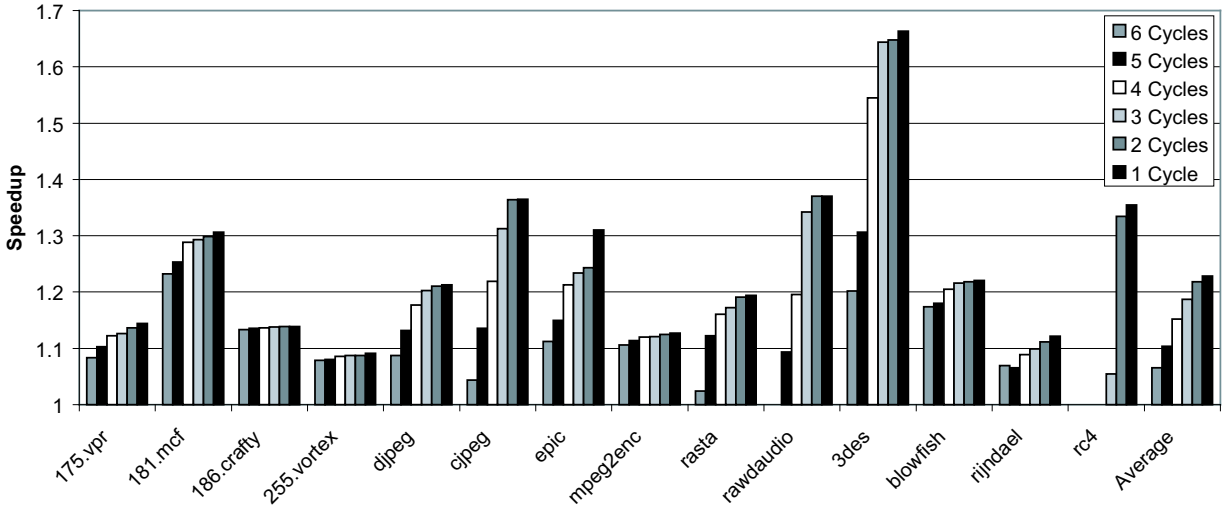



**Figure 6: The effect of CCA latency on speedups**

at which the CCA instructions become the critical path because of their added latency. As the latency increases, benefits from vertically compressing the dataflow graph disappear. The speedups that remain are solely due to the additional parallelism provided by the CCA.

Figure 7 shows the breakdown of instructions executed by the processor. The combined grey and black portions of the bars represent the percent of dynamic instructions that were provided by the frame cache. The black portion of the bars represents the fraction of dynamic instructions that were executed on the CCA. When using retirement based replacement schemes, it is very important to achieve high coverage, since CCA instructions only appear in the instruction stream from the frame cache. On average, 91% of instructions came from the frame cache in our simulations. The static discovery/retirement based replacement scheme was able to replace 35% percent of the frame cache instructions (or 32% of the total dynamic stream) with CCA operations.

As expected, a larger fraction of replaced instructions generally leads to better attained speedups. For example, 3des and rawdaudio both have a high percentage of their instructions executed on the CCA, and they are among the applications with the highest speedups in Figure 5. However, there is not a one-to-one correspondence between CCA coverage and speedup. Since many replaced subgraphs may not appear on the critical path, their acceleration will only have a small impact on program execution time.

The final experiment is presented in Figure 8, comparing the three different discovery and replacement strategies on processor performance. The first strategy employs static offline pattern discovery and relies on a translation table in decode to replace instances in the instruction stream. The second strategy performs dynamic discovery and replacement in the fill unit of the frame cache. The third strategy is static discovery with replacement done in the fill unit of the frame cache. All three of these strategies were run using the depth 4 CCA. A translation table size of 32 was chosen for the static-translation table strategy, because previous work [27] showed that only marginal increases (<0.5%) in dynamic coverage are possible beyond 20 patterns.

The most apparent trend in the graph is that the static-translation table strategy typically does rather poorly when compared against the other two techniques. Investi-

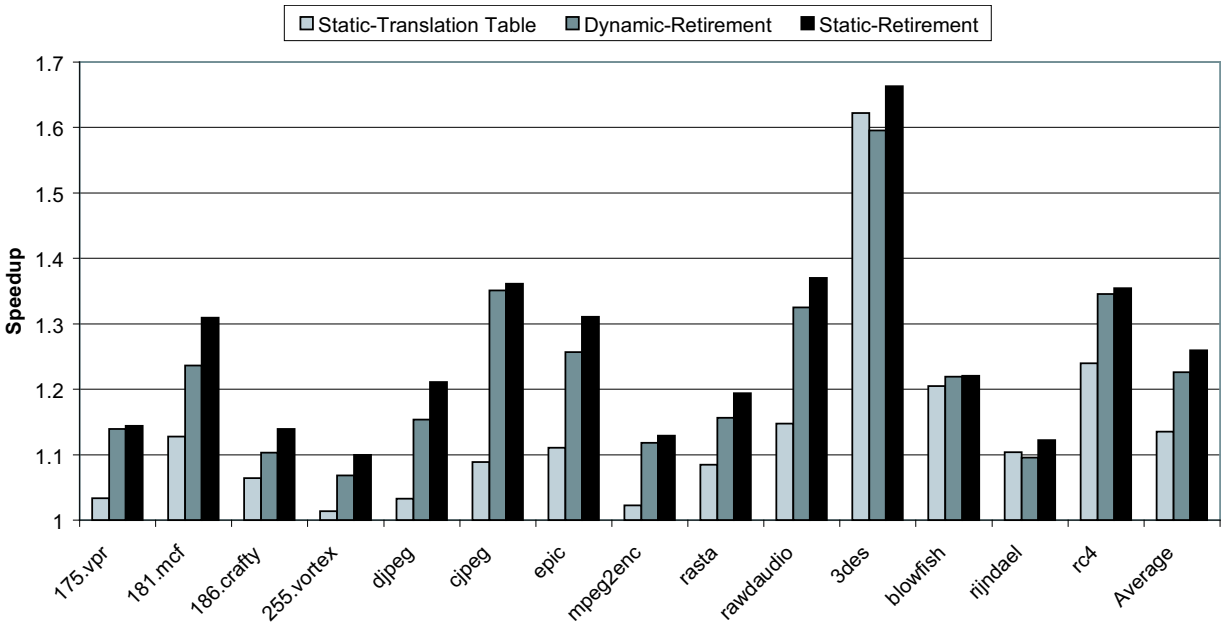



**Figure 8: The effect of various discovery/replacement strategies**

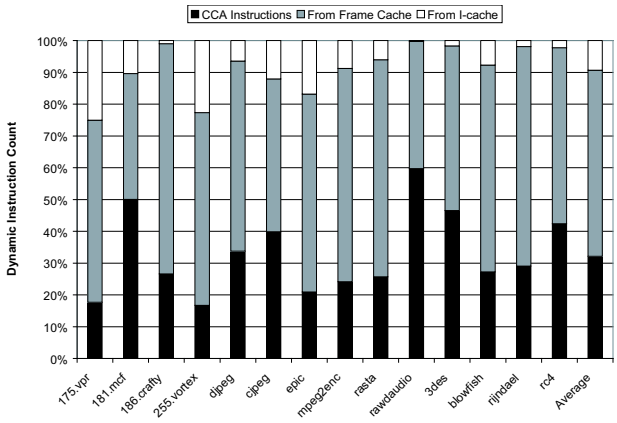



**Figure 7: Percentage of dynamic instructions from the I-cache and frame cache**

gation showed that this was not because of a limited number of available subgraphs. Rather, this method lacks a rePLay-style mechanism to roll back the processor state, which effectively allows subgraphs to span control flow boundaries. When any branch in a frame is mispredicted, an assertion occurs and the frame is discarded. Therefore, the frame can be treated as a large basic block for subgraph replacement. Without the rePLay mechanism, it is more difficult to allow subgraphs that execute on the CCA to span control flow boundaries. For this study, we conservatively do not allow any CCA subgraphs to span a branch. While this approach is correct, a large number of potential CCA subgraphs are lost with this method. Future work includes relaxing this constraint which will likely increase the effectiveness of the static-translation table.

The graph also shows that, as expected, the static discovery outperforms dynamic discovery with the frame cache. This is because the static scheme is using a much more powerful discovery technique than the simple dynamic heuristic. However, the dynamic heuristic does do quite well in a number of cases: 175.vpr, cjpeg, and rc4. One reason for this is the underlying ISA. Since the ARM ISA has only 16 architecturally visible registers (and several are reserved), the compiler often inserts a large number of loads and stores into the code for spilling. Since the CCA cannot execute memory operations, the spill code artificially limits the amount of computation in the dataflow graph. Larger amounts of computation generally results in more options during subgraph discovery, implying that the dynamic discovery algorithm is more likely to have its sub-optimality exposed. The difference between static and dynamic discovery strategies is likely to be more pronounced with an ISA that supports a larger number of registers and thus exposes more of the true data dependencies.

# 6. CONCLUSION

We have presented a novel mechanism to accelerate application performance by transparently customizing the instruction set of a processor. A configurable compute accelerator, which is a group of function units connected in a matrix-like configuration, is added to a general-purpose core to implement custom instructions. Subgraphs from a stream of processor instructions are identified and mapped onto this CCA. Two schemes are described for the identification of subgraphs: a dynamic one where subgraphs are identified on frames at runtime, and a static scheme where subgraph identification is done offline by the compiler. As part of the static scheme, we also proposed a novel technique to non-intrusively convey the subgraphs to the processor in order to control the CCA configuration at run-time. Two subgraph replacement schemes are discussed: one that occurs in the fill unit of a trace cache, and one that utilizes a statically loaded translation table during instruction decode.

Our experiments reveal that significant speedups are possible for a variety of applications, both from the embedded and general-purpose computing domains. The speedup was up to 66% for a 4-deep CCA (26% on average), and the area overhead is reasonably small. The

CCA has a moderate degree of latency tolerance, and thus can be more easily integrated into any modern processor pipeline.

## 7. ACKNOWLEDGMENTS

The authors would like to thank Brian Fahs, Greg Muthler, Sanjay Patel, and Francesco Spadini for the enormous help they provided in loaning/teaching us their simulation environment. Assistance in synthesizing the CCA was provided by Tae Ho Kgil and Seokwoo Lee. Additional thanks go to Stuart Biles, who helped shape the ideas presented through insightful conversations, and to Michael Chu and the anonymous referees who provided excellent feedback. This research was supported in part by ARM Limited, the National Science Foundation grants CCR-0325898 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

## 8. REFERENCES

[1] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *40th Design Automation Conference*, June 2003.

[2] P. M. Athanas and H. S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, 11(18), 1993.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of 2000 PLDI*, pages 1–12, 2000.

[5] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, pages 262–269, 2002.

[6] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA FPGA*, pages 141–150. ACM Press, 2001.

[7] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. Piperench implementation of the instruction path coprocessor. In *Proc. 33rd Intl. Symposium on Microarchitecture*, pages 147–158, Dec. 2000.

[8] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *36th Intl. Symposium on Microarchitecture*, pages 129–140, Dec. 2003.

[9] M. L. Corliss, E. C. Lewis, and A. Roth. Dise: a programmable macro engine for customizing applications. In *Proceedings of the 30th ISCA*, pages 362–373. ACM Press, 2003.

[10] J. Dehnert et al. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of CGO*, pages 15–24, Mar. 2003.

[11] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6), June 2001.

[12] B. Fahs et al. Performance characterization of a hardware mechanism for dynamic optimization. In *Proc. 34th Intl. Symposium on Microarchitecture*, pages 16–27. IEEE Computer Society, 2001.

[13] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. 31st Intl. Symposium on Microarchitecture*, pages 173–181, Dec. 1998.

[14] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES*, 2003.

[15] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1997.

[16] I. Huang and A. M. Despain. Synthesis of Application Specific Instruction Sets. *IEEE TCAD*, 14(6), June 1995.

[17] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the The 5th HPCA*, page 125. IEEE Computer Society, 1999.

[18] P. Lowney et al. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[19] G. Lu et al. Morphosys: A reconfigurable processor trageted to high performance image application. In *Proceedings of the 11th IPPS/SPDP'99 Workshop*, pages 661–669, 1999.

[20] P. Metzgen. A high performance 32-bit alu for programmable logic. In *Proceeding of the 2004 ACM/SIGDA 12th FPGA*, pages 61–70. ACM Press, 2004.

[21] T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–11, 1998.

[22] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Trans. Comput.*, 50(6):590–608, 2001.

[23] A. Peymandoust et al. Automatic instruction set extension and utilization for embedded processors. In *14th ASAP*, June 2003.

[24] J. Phillips and S. Vassiliadis. High-Performance 3-1 Interlock Collapsing ALU's. *IEEE Trans. Comput.*, 43(3):257–268, 1994.

[25] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Function Units. In *Proc. 27th Intl. Symposium on Microarchitecture*, pages 172–180, Dec. 1994.

[26] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th Micro*, pages 238–247. IEEE Computer Society, 1996.

[27] F. Spadini, M. Fertig, and S. J. Patel. Characterization of repeating dynamic code fragments. Technical Report CHRC-02-09, CHRC University of Illinois at Urbana-Champaign, 2002.

[28] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *Intl. Conference on Computer Aided Design*, Nov. 2002.

[29] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[30] Z. A. Ye et al. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th ISCA*, pages 225–235. ACM Press, 2000.

[31] 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 ISCA*, 2004.

[32] P. Yu and T. Mitra. Characterizing Embedded Applications for Instruction-Set Extensible Processors. In *41st Design Automation Conference*, June 2004.