An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors

Nathan Clark[1], Jason Blome[1], Michael Chu[1], Scott Mahlke[1], Stuart Biles[2] and Krisztián Flautner[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{ntclark, jblome, mchu, mahlke}@umich.edu

[2]ARM, Ltd.
Cambridge, United Kingdom
{stuart.biles, krisztian.flautner}@arm.com

ABSTRACT

Instruction set customization is an effective way to improve processor performance. Critical portions of application dataflow graphs are collapsed for accelerated execution on specialized hardware. Collapsing dataflow subgraphs will compress the latency along critical paths and reduces the number of intermediate results stored in the register file. While custom instructions can be effective, the time and cost of designing a new processor for each application is immense. To overcome this roadblock, this paper proposes a flexible architectural framework to transparently integrate custom instructions into a general-purpose processor. Hardware accelerators are added to the processor to execute the collapsed subgraphs. A simple microarchitectural interface is provided to support a plug-and-play model for integrating a wide range of accelerators into a pre-designed and verified processor core. The accelerators are exploited using an approach of static identification and dynamic realization. The compiler is responsible for identifying profitable subgraphs, while the hardware handles discovery, mapping, and execution of compatible subgraphs. This paper presents the design of a plug-and-play transparent accelerator system and evaluates the cost/performance implications of the design.

1. INTRODUCTION

Application specific instruction processors, or ASIPs, have the potential to meet the challenging high-performance demands of embedded applications. Integrating specialized hardware computation blocks is a pervasive technique used to accelerate individual applications. This specialized hardware is exploited through the use of customized instructions or instruction set extensions, which allow critical portions of an application's dataflow graph to be more efficiently executed than through the general purpose processor. Several commercial tool chains design ASIPs with customized instructions, including ARM OptimoDE, Tensilica Xtensa, and ARC Architect. ASIP solutions are desirable because they offer a cost and energy efficient approach to increasing processor performance. Further, they maintain a degree of system programmability; this substantially reduces the software burden of porting applications to new platforms by providing the flexibility of a processor-based solution.

The central problems with an ASIP approach are the hardware design and software migration time/costs. ASIP designs incur substantial non-recurring engineering costs. For example, each new ASIP must be verified both from the functionality and timing perspectives. Additionally, a new mask set must be created to fabricate the chip. On the software side, the compiler must be retargeted to each new processor and any hand-written libraries must be migrated to the new platform. Automation of some of these tasks may be possible; however, the majority of this work is still a manual process. All of these challenges make it difficult to adopt a new ASIP despite the potential advantages.

The difficulties with ASIP design are altogether avoided in general-purpose processors (GPPs) by offering a small family of processors with the same instruction set and software toolchain. The disadvantage of this approach is poor performance. Efficiency gains achieved through specialization are not possible, as one processor is created for all application domains. In this paper, we investigate the use of application specific processing technologies in the context of a GPP. More specifically, the objective is to make use of specialized hardware blocks and custom instructions while maintaining a fixed processor design and general-purpose instruction set. We refer to this technique as *transparent instruction set customization*. Transparent instruction set customization on a GPP enables many of the performance advantages of ASIPs without the associated overhead.

Our work builds upon prior work in which a configurable compute accelerator (CCA) is defined to execute selected computation subgraphs [7]. The CCA consists of an array of simple function units interconnected in a feed-forward manner. The CCA is exploited using one of two methods. First, a dynamic method was proposed to identify and remap subgraphs to the CCA in a trace cache fill unit [21]. Second, a static strategy identifies subgraphs offline during compilation and replaces the subgraphs with CCA instructions at run-time using a translation table provided in the binary. Both solutions have major drawbacks. The dynamic approach relies on a trace cache and its associated hardware optimization system. Such hardware is generally not appropriate for embedded processors due to cost and energy consumption. Further, run-time identification of patterns is inherently constrained to simple approaches as it is performed during application execution. The static approach offers no flexibility in terms of supporting multiple accelerators, as a fixed mapping to the CCA is assumed. Further, register encoding limitations in the GPP instruction set severely restrict the size of subgraphs that can map to the CCA.

This paper presents the design of an architectural framework to efficiently support transparent instruction set customization in an embedded GPP, such as an ARM. The framework utilizes a hybrid approach of statically-identified, dynamically-realized custom instructions. Subgraphs tar-

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

geted for acceleration are identified during compilation or as a post-link optimization and are marked in the program executable. At run time, subgraphs are discovered, mapped, and executed on specialized hardware blocks. The hybrid approach enables the combination of sophisticated offline subgraph detection algorithms with the flexibility of online realization of the customized instructions.

Several important challenges are addressed in the proposed framework. First, a plug-and-play accelerator model is defined that consists of an augmented GPP pipeline with a predefined interface to an optional hardware accelerator block. The augmented GPP is designed and verified once. Second, the framework supports a wide range of accelerator designs including standard predefined accelerators (such as a CCA) and user-defined hardware accelerators. Regardless of the specific accelerator (or lack thereof), a single application binary is created and executed on all platforms. Third, the acceleration of complex acyclic computation subgraphs is supported. Prior work often limits subgraphs to linear chains, thereby precluding many of the performance benefits achieved with custom instructions in ASIPs. Fourth, the limited expressibility of the target instruction set architecture in terms of register names does not limit contents of the selected subgraphs. For GPP instruction sets such as ARM with only 16 registers, register spills often limit subgraphs to small sizes, thus its important to overcome this limitation. Finally, a low-cost and energy-efficient solution is selected to make the approach appropriate for embedded computing.

2. RELATED WORK

Utilizing instruction set extensions to improve the computational efficiency of applications is a well studied field. Examples of industry standard domain specific instruction set extensions such as Intel's SSE or AMD's 3DNow! multimedia instructions are commonplace in modern systems. Techniques for generating domain specific extensions are typically ad-hoc, where an architect examines a family of target applications and determines what extensions can be expected to provide increased performance.

In contrast to domain specific extensions, many techniques for generating application specific instruction set extensions have been proposed [3, 6, 8, 12, 13, 25]. Each of these algorithms provide either exact formulations or heuristics to effectively identify those portions of an application's dataflow graph that can efficiently be implemented in hardware. These techniques are not directly applicable to this work, because they do not take into account the underlying structure of the execution hardware.

Much attention has been given to the structure of a CCA design for accelerating dataflow subgraphs. The research in [27] proposed using a fine granularity CCA based on slightly specialized FPGA-like elements. Restricting the interconnect of the FPGA-like elements reduces 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 significant drawbacks of a large number of control bits and the substantial delay of FPGA-like elements. A key observation is that the flexibility of an FPGA is generally more than is necessary for dataflow graph acceleration.

Other recent work [5, 23] proposed CCA structures specifically optimized for linear chains of execution. That is to say these structures only execute subgraphs that have two inputs, one output, and a small number of intermediate nodes. Constraining the subgraphs in this way has been shown to effectively increase the bandwidth of execution resources, however it severely restricts the performance increase from dataflow graph compaction [28]. Since embedded processors are typically in-order and single-issue, resource contention is not a major issue. In this work, we use a more generic CCA architecture, similar to [7], to support the execution of more arbitrary acyclic dataflow subgraphs. These subgraphs are larger than simple linear subgraphs, and attack the computation limitations of embedded processor instead of resource limitations.

Once a CCA execution engine is developed, techniques are needed to map dataflow subgraphs onto the execute engines. Many hardware based frameworks exist for this process. Most of these 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 (on the order of 10,000 cycles [11]). That is, once instructions retire from the pipeline and a trace is constructed, there is ample time before that trace will be needed again. Two recently proposed schemes [7, 23] used this latency to perform the mapping of dataflow subgraphs onto specialized execution hardware. While the trace cache provides an excellent place for mapping subgraphs into the instruction stream, it is far too large and inefficient for embedded domains.

A simplified dynamic subgraph mapping system was described in [15, 24]. These papers used the design proposed in [22] as the baseline of their system, which greatly simplifies the mapping problem. Because our goal was to allow for more flexibility than their CCA design allowed for, our presented identification algorithm is much more complex.

Other recent work [5] proposes using the DISE [9] framework to dynamically replace subgraphs in the instruction stream. A special instruction is used to signal the DISE engine, which then inserts the appropriate control logic into the pipeline. This model requires a DISE aware operating system and processor, since the subgraphs are specified in the binary at load time, and must be replaced to execute the binary at runtime. Conversely, the framework proposed in this work does not affect the operating system, nor does it require any special replacement engine to run the binary.

The key difference between this paper and prior work is that instead of proposing a CCA *architecture*, we propose an *architecture framework* into which many CCA architectures fit. This framework provides a clean interface between a processor pipeline and a CCA, enabling easy customization of a CCA for the expected system workload. We demonstrate how the framework can process a dataflow subgraph to generate CCA instruction on the fly, without the costs associated with a trace cache. Beyond the architecture framework, we describe the compilation process, by which subgraphs are identified in applications and communicated to the architecture framework.

3. ARCHITECTURAL FRAMEWORK

The primary contribution of this work is a configurable architectural framework to facilitate transparent instruction set customization. This framework allows architects to design hardware accelerators tuned for an expected workload and easily incorpoorate them into a general purpose pro-

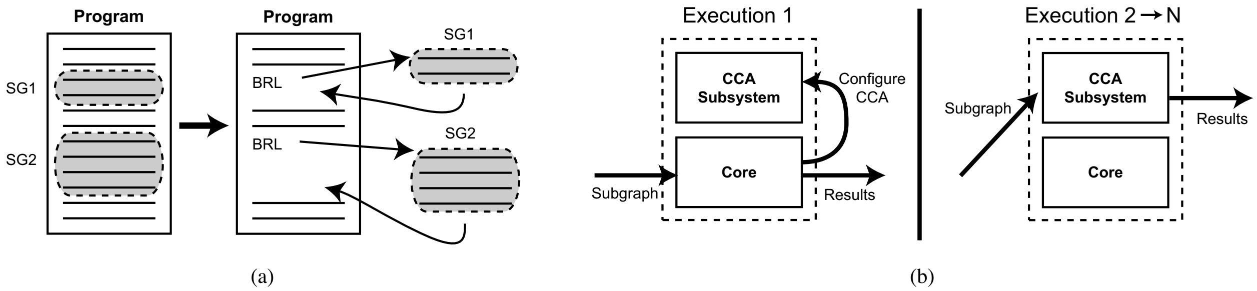


Figure 1: A high-level overview of the executing with a CCA: (a) subgraph identification and relocation and (b) setting up the CCA subsystem on the first invocation of a subgraph for future uses

cessor via a well-defined interface. The use of a workload-specific accelerator allows manufacturers to build machines targeted toward many domains at the cost of designing and verifying only a single general purpose core and a set of applicable accelerators.

This section begins with an operational overview of the framework. The remaining subsections present a description of the proposed pipeline microarchitecture, the stages of execution of dataflow subgraphs within this pipeline, and the system interface to support subgraph execution on the system.

3.1 Overview

The objective of a framework for transparent instruction set customization is the support of a hybrid form of execution where subgraphs are statically identified and dynamically realized. Static identification refers to offline compiler identification of potential subgraphs for execution on custom hardware. Dynamic realization refers to hardware synthesizing the custom instructions at run-time and offloading their execution to the CCA.

The high-level process is illustrated in Figure 1. Initially, a program is analyzed by the compiler to identify critical computation graphs that can be mapped onto the CCA. The operations that comprise the subgraphs are pulled out of their original locations and placed into a separate function body as illustrated in Figure 1(a). The BRL, or branch and link, instruction is used to denote a function call in this figure. Dynamic realization is accomplished in two phases. Initially, the subgraph is executed on the hardware of the uncustomized core, denoted as Execution 1 in Figure 1(b). During this execution, a hardware engine determines the CCA configuration necessary to execute the entire subgraph as an atomic unit. In essence, a complex opcode is synthesized on the fly. On subsequent executions of the subgraph, the new complex opcode is substituted for the invocation of the subgraph function. Thus, as shown in Figure 1(b), the standard hardware must execute the first occurrence of the subgraph, while all subsequent executions will be relegated to the CCA.

The combination of static identification and dynamic realization enables powerful offline algorithms to optimize code for subgraph extraction. Further, a well-defined architectural interface introduces a layer of flexibility so that previously designed and verified cores can be easily integrated with multiple CCA designs. The remainder of this section expands the details of the architectural framework to accomplish this model of execution.

3.2 Pipeline Organization

Figure 2 presents a block diagram of the proposed architecture framework. The baseline processor, at the bottom of the figure, is augmented with the CCA subsystem at the top of the figure. The CCA subsystem consists of three major parts: the CCA itself, a configuration cache, and a control generator. The control generator is responsible for examining a sequence of retiring instructions and determining the required control signals for the CCA. Each entry of the configuration cache specifies the necessary control signals for configuring the CCA, including the opcode implemented on each CCA function unit, the interconnect between function units, and any literal values used by the subgraph.

The core processor is augmented in several places to interact with the CCA. Changes primarily occur in the instruction fetch stage of the pipeline, where instruction stream substitution occurs. The branch target address cache, or BTAC (sometimes called BTB in other literature), is extended to store additional information to decide when it is possible to substitute a CCA instruction for an invocation of a subgraph function. To accomplish this, a CCA configuration cache entry and register indexes for values consumed by the subgraph are included in the BTAC. The decode and writeback stages are also modified to provide register inputs and accept register results from the CCA.

Central to the framework is a well-defined interface between the core and the CCA subsystem. The interface is designed so that the core can use multiple CCA designs. Since any hardware placed on the CCA subsystem increases the cost of customization, the necessary structures were integrated into the main pipeline as much as possible while maintaining the flexibility of the interface. The numbered arrows in Figure 2 denote the five interface points between the CCA subsystem and the CPU. These points are the only communication required between the CPU and the CCA subsystem:

1. The CCA subsystem generates entry information for the BTAC. This includes subgraph live-in register indexes and a configuration cache index where the control bits are stored.
2. During instruction decode, the configuration cache index is sent to the CCA subsystem.
3. As previously mentioned, the decode stage also provides the CCA with values for registers that are inputs to the subgraph.
4. The output values from the subgraph are relayed from the CCA subsystem back to the CPU for register writeback.

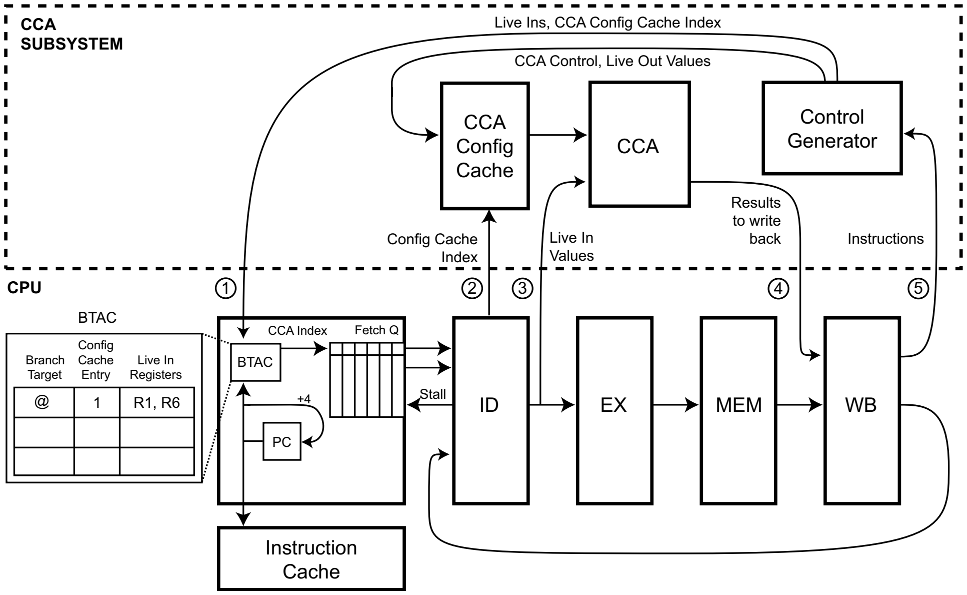


Figure 2: Transparent instruction set customization architectural framework

5. After retirement, completed instructions are provided to the control generator so that it can synthesize the CCA instructions from dataflow subgraphs.

3.3 Dataflow Subgraph Execution

A single instruction is added to the baseline instruction set to allow the compiler to delineate patterns for execution on the CCA hardware. A discussion of how the compiler uses these instructions follows in Section 4. The introduced instruction is dubbed BRL' because its semantics are very similar to a branch-and-link operation commonly used for subroutine calls. BRL' is treated just like a normal branch-and-link instruction in processors without a CCA subsystem: the current program counter (PC) is stored to a link register and control branches to the branch target address. The processor without a CCA will execute the instructions in the target subroutine and return to the call site, just as it would for any other subroutine. To a processor with a CCA subsystem, the BRL' signifies the start of a subgraph to execute on the CCA.

When the BRL' is fetched from the instruction cache, its address is used to index into the BTAC. The BTAC is a standard component of modern branch prediction schemes used to hold the destination of a taken branch. In this framework, the BTAC is augmented to contain two additional pieces of information for each BRL' instruction. Register numbers for the inputs to CCA instructions are one of the additional pieces of information. These values are fed to the instruction decode stage for register reads. An index into the CCA configuration cache is the second additional piece of information stored in the BTAC. The configuration cache on the CCA subsystem contains the control bits for the CCA execution unit. If a BRL' hits in the BTAC, the configuration cache index is passed through the pipeline with other control bits and the PC simply increments to the next instruction (i.e., the branch is not taken because the BRL' was recognized as a subgraph). This prevents pipeline bubbles that would form if the branch target was taken. If the BRL' misses in the BTAC, then it is executed as a normal BRL and control branches to the procedure.

Recall that control bits from the BTAC provide the registers that are read during the decode stage of execution. Since we assume only two register reads are supported in one cycle, it may be necessary to use multiple cycles to read all of the operands necessary for the CCA instruction. Extra communication is provided allowing the decode stage to stall the fetch unit in order to facilitate this multi-cycle register read. As the registers are read, they are passed to the CCA system, keeping the width of the interface connection to a minimum.

The BTAC also passes a configuration cache index through the decode stage and into the CCA system. The configuration cache contains information pertaining to the routing of the signals on the CCA, as well as the operations to perform at each node in the CCA grid. This information is separated from the BTAC for two main reasons. First, the number of control bits is highly dependent on the structure of the CCA. Putting the configuration cache in the core, as part of the BTAC, effectively restricts the size and organization of the CCA, since the number of control bits is set a priori. Second, putting the control bits in a separate configuration cache allows reuse of the same control bits for different subgraphs. For example, if two separate subgraphs were identical except for the registers that provide their inputs, they could share an entry in the configuration cache.

Once the registers and configuration data are passed along, the CCA executes the subgraph as a single operation and feeds the results to the writeback stage of the core. The CCA operates like any other function unit in this regard. An example of a potential CCA implementation can be seen in Figure 3. The CCA here is implemented as a grid-like grouping of function units with full interconnect between adjacent rows. Because of delay constraints, the two rows have slightly different opcodes available for execution, the white nodes support add, subtract, compare, sign extend, and all logical operations, while the gray nodes only support sign extend and logical operations. The design in this figure was taken directly from our previous work [7], and a

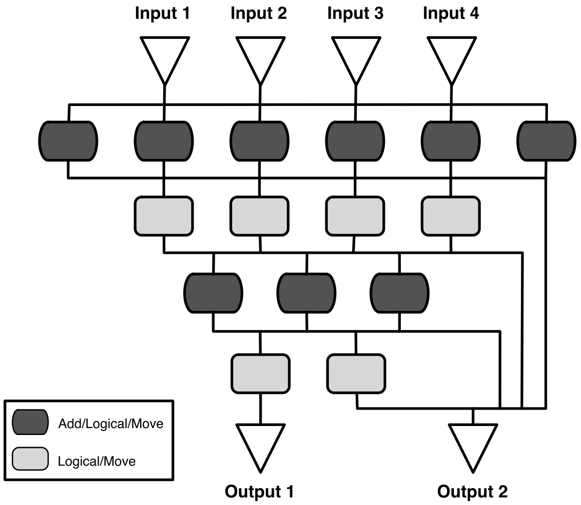


Figure 3: Example of a CCA implementation

more thorough discussion of the design rationale is described there. After execution on the CCA, results are written to the register file and instructions are fed the the CCA control generator, which is responsible for mapping subgraphs onto the CCA.

3.4 Dataflow Subgraph Control Generation

Dynamically determining the control signals for the CCA is the most complex portion of the CCA subsystem, and is best illustrated through an example, as shown in Figure 4. In this example, the subgraph in the top left corner will be mapped to the CCA in the bottom left corner. The nodes of this CCA are labeled A-O for easy reference. The assembly code and subgraph in this example were taken from the Rijndael encryption algorithm.

Instructions are fed through the control generator one at a time after the writeback stage. The two loads at the top of the example are fed through and ignored, since they are not part of a subgraph. When the third instruction, a BRL', is retired, it signals the beginning of a subgraph and that the CCA subsystem should generate control information for it. The PC of the BRL' is recorded so that it can be used to update the BTAC with the appropriate data when the subgraph has been fully processed.

After the BRL', instructions are mapped to the CCA grid as they enter the control generator. Determining where to map the instructions requires several pieces of state, shown in the right portion of Figure 4. The table at the top of each step is a content addressable memory, or CAM, that maps a stack offset to a node that produces the value. The CAM is used to determine which node in the CCA produced the spilled value when a different operation in the gets its input from the stack. This allows the control flow generator to eliminate spill code of transient values within the subgraph. The size of this CAM equals the number of nodes in the CCA, since each node could potentially spill its produced value.

Since the proposed CCA subsystem does not support memory access operations, if the compiler is unable to allocate registers to all of the transient variables in a subgraph, then spill code would effectively partition the subgraph. This restricts performance improvement simply because of register pressure and is our rationale for performing spill code elimination.

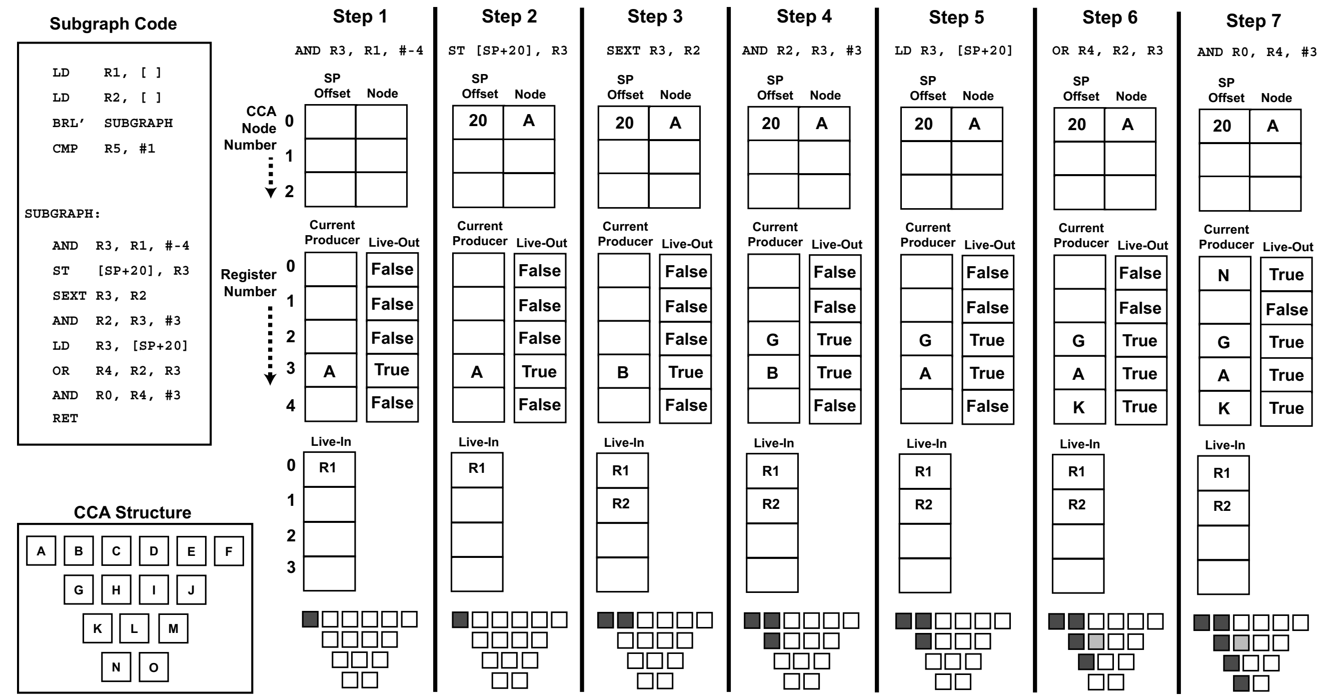
The second piece of state in Figure 4 is the current producer table. For each register in the machine, this table contains the node of the CCA that produced the most recent value computed for that register. The control generator also keeps two tables marking live-in and live-out values of the current subgraph. The table of live-out values records every time a value is produced by a CCA node. It is necessary to assume that all register values created are live-out, and must be written to the register file, until proven otherwise. The live-in registers record which registers are needed as inputs to the subgraph and are communicated to the BTAC after control generation is complete. The live-in table is the size of the maximum number of inputs allowed on the CCA execution unit, in this case four. At the bottom of each step is a running count of the nodes in the CCA (marked in dark gray) which have been allocated an operation by the control generator.

When the first instruction, AND R3, R1, #-4, enters the control generator, that instruction looks up each source operand in the current producer table. Since R1 has no current producer, it is added to the list of live-ins. No other nodes in the subgraph create results that this operation consumes, so the AND instruction can be assigned to node A in the first row of the CCA. The current producer table is updated to reflect that R3 is generated by CCA node A, and R3 is marked as potentially live-out. The opcode AND and constant -4 are stored as the function executed by node A. The state after processing the AND instruction is reflected as Step 1 in Figure 4.

Spill code for R3 is the next instruction entering the control generator. The compiler guarantees that any spill code within the subgraph is only for transient values, and thus can be optimized away without affecting the correctness of the program. In this example the spill code stores R3 to stack offset 20. Since R3 is produced by node A, that node value is stored with an index of the stack offset in the CAM. Future instructions that use values spilled on the stack, use the CAM to determine which node in the CCA generates the instruction's inputs. Step 2 in Figure 4 shows the control configuration state after mapping the store instruction.

Following the spill instruction, the SEXT instruction enters the control generator. Since this instruction uses R2, and R2 has no producer in the current producer table, R2 is marked as live-in and the instruction is placed at node B in the first row of the CCA. This instruction produces a value for the spilled register R3, so the current producer of R3 is changed to node B, and the live-out bit of R3 remains set. When the next AND instruction is mapped, a look up of its source operand R3 shows that node B produces it. This means that the AND operation must be placed in the row below node B, in this case node G. The current producer table is then updated to reflect that R2 is now produced by node G.

The next retired instruction is the spill code load for R3. The control generator looks up the spill offset in the CAM and finds that node A generated the value being loaded. Thus, the LD instruction resets the current producer of R3 to node A, and it remains marked as live-out. After the spill code load, an OR instruction with sources R2 and R3 is processed. Both of these sources are produced by other nodes in the subgraph. Since it is dependent on node G,



```
LD    R1, [ ]
LD    R2, [ ]
BRL'  SUBGRAPH
CMP   R5, #1

SUBGRAPH:
AND   R3, R1, #-4
ST    [SP+20], R3
SEXT  R3, R2
AND   R2, R3, #3
LD    R3, [SP+20]
OR    R4, R2, R3
AND   R0, R4, #3
RET
```

Figure 4: Example mapping subgraph onto a CCA

this operation must execute in the third row. It is placed at node K and updates the current producer table accordingly. In addition, because the operation requires a source from the first row (R3), a move must be inserted in row 2. Moves are necessary because only adjacent rows in this CCA architecture are directly interconnected. This move is marked in light gray at the bottom of step 6. Similar to previous instructions, the AND is inserted in the last row of the CCA. The final instruction, an RET, marks the end of this pattern.

Once the end of the subroutine is reached, the control data is not yet ready to be written to the BTAC and CCA configuration cache, since there exist more live-outs than are supported by the execution system. The compiler is responsible for proving that only a limited number of live-outs exist in each pattern. Therefore, to determine which ones are not actually live-out, it is necessary to monitor the retiring instruction stream and unset the live-out bit for any register that is defined before used.

Determining true live-outs can either be done by waiting for other instructions to naturally kill potential live-outs, or by having the compiler insert artificial instructions to ensure that false live-outs are killed quickly. Regardless of the strategy, the latency of killing live-outs should prove irrelevant to system speedup as prior work [11] has shown that moderate latencies are likely between trace retirement and recurrence.

If at any point the control generator cannot map a subgraph onto the underlying CCA execution unit, then it simply aborts control generation for that pattern. This allows applications compiled for CCA subsystem 1 to run on CCA subsystem 2 even when the second may not support all the subgraphs that the first supports. Providing the dynamic control generator as part of the CCA subsystem is key to the retargetability of the system.

To determine the overhead of adding the CCA control generator, we synthesized a control generator to target the CCA execution unit shown in Figure 3. Using Synopsis design tools, with a 250 MHz target clock speed, and a 130nm Artisan standard cell library, we found that the total die area of the control generator is only 0.169mm^2. For comparison, the die area of an ARM 926 processor in 130nm is 5.0mm^2. The delay of the control generator is is only 0.46ns plus latch setup time, meaning that this can easily be fit into a modern embedded processor's timing model.

4. COMPILER CODE GENERATION

In order to exploit the specialized CCA hardware, a CCA cognizant compiler requires several new steps in the code generation process. The overall structure of the compiler flow is shown in Figure 5; steps added for CCA compilation are gray in this figure. Normal code compilation has three major steps: scheduling, register allocation, and postpass scheduling of spill code. At the beginning of compilation, a CCA compiler must determine which dataflow subgraphs should execute on the CCA. The remaining complexity of compiling for a CCA stems from the fact that some phases of compilation need to treat the subgraphs as atomic units and other phases need to understand each constituent node of the subgraph. Each of these phases is explained in detail in the remainder of this section.

4.1 CCA Compiler flow

Subgraph Identification: Given a dataflow graph as input, subgraph identification determines which portions should be executed on the CCA. This is very similar to the problem of technology mapping in VLSI design. In the general case, where the subgraphs are not necessarily trees, the problem is NP-hard [1]. Difficulty of the problem is the primary reason subgraph identification is performed at compile time instead

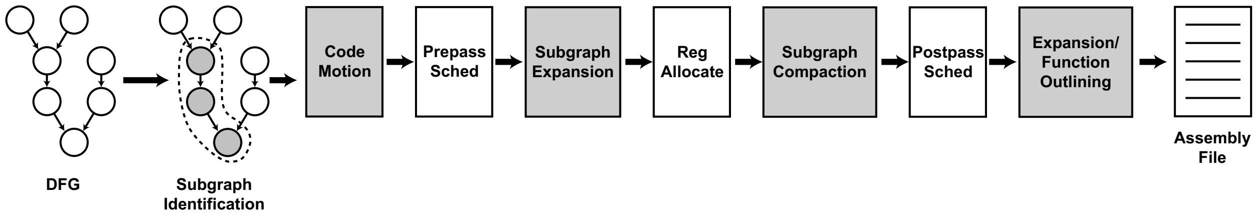


Figure 5: Compiler flow diagram. New steps in the compilation process are shown in gray.

of runtime.

Heuristics for solving subgraph identification have been the subject of much related work [1, 19, 20]. Because this is a very complicated issue, the specific details of our algorithm fall outside the scope of this paper, and we refer the curious reader to our previous work [8], which gives a more thorough treatment to the topic.

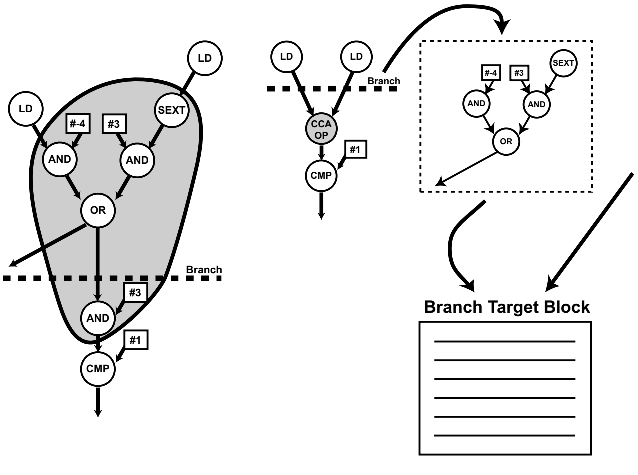
From a high level, subgraph identification is performed in two steps. First, subgraphs are enumerated within a basic block or superblock, using a branch and bound algorithm. This algorithm generates the set of all subgraphs capable of being executed on the target CCA. In the case that a block is too large for full enumeration, the block is intelligently split into smaller pieces, each of which is fully enumerated.

After enumeration, the second step of subgraph identification is selecting which of the enumerated subgraphs to execute on the CCA. At issue is that each operation in the dataflow graph may appear in multiple subgraphs, yet each operation can only be mapped onto the CCA as a member of one subgraph. Thus, it is necessary to either replicate operations or a subset of subgraphs must be selected to maximize performance subject to the constraint that each operation appear in only one subgraph. Beyond that, it is also necessary to determine if the target CCA is capable of executing the subgraph more efficiently than the constituent operations on the baseline processor. For example, if a subgraph consists of two dependent ADD operations, and the latency of the target CCA is three cycles, then executing that subgraph on the CCA is not worth the overhead. In this work, subgraph selection is accomplished using a dynamic programming heuristic described in [8].

It is important note that subgraph identification is performed before register allocation. Performing subgraph identification after register allocation introduces many false dependencies within the dataflow graph, and hinders the size of the subgraph that can be discovered. Indirect evidence of these dependencies exists in the effectiveness of register renaming logic in superscalar processors. Even though false dependencies are a major problem, most related work performed subgraph identification after register allocation, so it could be done at link-time or run-time.

Code Motion: After subgraph identification, the selected subgraphs are collapsed into a single instruction. In order to effectively mark subgraphs as a special procedure calls for the hardware, it is essential that the scheduler maintain the instruction order such that the subgraphs appear contiguously in the code. Collapsing the subgraph into a single node cleanly prevents operation reordering without altering the scheduler internals.

When collapsing the subgraph, a problem arises if the subgraph crosses branch boundaries. Previous work has shown that preventing subgraphs from crossing branch boundaries greatly constrains the size of the subgraphs [28]. Thus, a decision must be made as to where to insert the CCA node relative to the crossed branch. We consider the two extreme possibilities: before the first branch and after the last branch. Both choices have ramifications which must be corrected in the code. The process of placing a CCA operation after the last branch boundary is termed *downward code motion* and placing the CCA operation before the first branch boundary is termed *upward code motion*. Without loss of generality, each form of code motion is considered for a single branch operation.



(a) (b)

Figure 6: The process of downward code motion as (a) the cross branch subgraph is identified and (b) code is replicated in a new block

In downward code motion, the subgraph is assumed to span the not taken direction of the branch. The problem that arises is there could potentially be portions of the collapsed subgraph which need to be executed before the code at the branch target is executed. Consider the example in Figure 6(a), which is a portion of the dataflow graph from the Rijndael encryption benchmark. The subgraph identified for collapsing is encircled in gray. If the collapsed node is executed after the branch boundary, the application will execute correctly as long as the branch is not taken. However, if the branch is taken, then there are operations within the collapsed subgraph that did not execute but should have. These are the operations from Figure 6(a) that are within the encircled gray subgraph and above the dotted branch line.

After placing the collapsed subgraph below the branch boundary, the portion above the branch must be replicated. Figure 6(b) shows this process when the branch target is a block of code with a multiple entries. A new block is created with the code region from the collapsed node. This code region then unconditionally branches to the original branch target. In the case where the target block has only one control flow entry point, this new block is simply collapsed into the beginning of the branch target block. This process is essentially the same as the bookkeeping code induced through downward code motion used during trace scheduling [10].

Downward code motion easily extends to patterns which cross multiple branch boundaries. Generally speaking, executing subgraphs that cross branch boundaries increases the size of the computation subgraphs executed on the CCA, which improves performance. The trade off is increased code size from operation replication.

The alternative to downward code motion is to place the collapsed subgraph above the branch boundary, or upward code motion. In this case, the CCA could potentially execute code that should never have been executed, and therefore speculates that the branch will not be taken. If the branch is taken, then code must be inserted to repair the incorrectly executed instructions. Additionally, operations that could potentially cause exceptions, such as a divide or load operations, must not be speculatively executed to guarantee correct execution.

The CCA compiler system implemented in this work exclusively uses the downward code motion process, placing the CCA operation after the branch. This method always produces functionally correct code regardless of excepting instructions.

One potential area where downward code motion has difficulty is if a value produced by a CCA instruction is consumed by the branch. For example, in Figure 6(a), if the live out from the OR operation was used to determine whether or not the branch is taken, then this subgraph cannot be moved below the branch. In this case, the CCA compiler rejects this potential subgraph as a target for collapsing.

Prepass/Postpass Scheduling: These two phases of compilation are unchanged from the standard compiler. Later in compilation, the subgraphs are turned into special function calls using the BRL' instructions, and thus, it is important to keep all of the subgraph instructions contiguous in the schedule. This is the main reason why subgraphs are compressed into atomic instructions.

Subgraph Expansion: While scheduling considers the subgraphs as atomic units, register allocation needs to consider each instruction separately in order to properly assign the registers to the internal values. Recall that processors without CCA subsystems must still be able to execute the code generated for processors with CCAs. This mandates that the subgraph must be register allocated. Without expanding the subgraphs, it is difficult for the register allocator to correctly construct live ranges and assign registers.

Register Allocation: Expanding the subgraphs before the register allocation allows this phase of compilation to be relatively unchanged. Registers are simultaneously assigned to all instructions, including the expanded subgraphs, just as they would normally be. The only change has to do with the addition of some caller save code for the subgraph. Recall that the subgraph will be implemented as a subroutine call using the BRL' instruction. The BRL' will overwrite the link register, if it is not saved to the stack. Thus, a save and restore of the link register are added on either side of the subgraph. No additional caller save code is necessary, since we know exactly which registers will be used in the subgraph and have already allocated appropriately. In calling conventions where the link register is already callee saved in the function prologue, this additional code is not necessary.

An optional optimization to register allocation is to intelligently prioritize the variables to be allocated. Since the CCA control generator is capable of collapsing spill code of transient values within subgraphs, there is no need to allocate a register for those values at the expense of other variables. Giving these transient values very low priorities, guarantees that register allocation will spill them if necessary, and effectively increases the number of registers available to the machine.

Subgraph Compaction: After register allocation, the full subgraph is again compressed to an atomic node in preparation for postpass scheduling. This process is complicated slightly by spill code that is introduced in relation to the subgraph. If a transient value in the subgraph is spilled, e.g. R3 in Figure 4, then this must be combined into the subgraph. By placing this spill code in the subgraph, the compiler guarantees that the results are not needed outside of the subgraph and these loads/stores can be optimized away by the CCA subsystem. If a value that is live-out of the subgraph is spilled and also consumed in the subgraph, then the store that spills the live out is replicated outside of the subgraph. A copy of the store must remain in the subgraph so that the control generator can determine which node produced the spill value. Once the subgraphs are compacted, postpass scheduling is performed.

Function Outlining: After postpass scheduling, the subgraphs are again expanded into their constituent nodes. Each subgraph is moved to a separate portion of the code and a BRL' is inserted at the former location of the subgraph. This process is referred to as *function outlining.*

The technique of function outlining (sometimes called procedure abstraction) has been used in previous work [16, 18] for code size reduction. Since groups of instructions are often repeated at several different places within an application, function outlining can combine these instances into one procedure. While the primary purpose of our function outlining is to delineate subgraphs for the hardware, it also provides us with code compression to help offset some of the code replication from subgraphs that cross branch boundaries. It should also be noted that the code size reduction could be improved by making the register allocator more proactive in assigning the same register values to isomorphic subgraphs.

With function outlining complete, an assembly file is output that can be run on any processor which recognizes the BRL' instruction.

5. EXPERIMENTS

Our experimental system was built on top of the Trimaran compiler infrastructure [26]. Trimaran was retargeted for the ARM instruction set and augmented with a parameterized subgraph matcher to recognize dataflow subgraphs that map onto the underlying CCA infrastructure. Once the subgraphs are identified, code motion, scheduling, and the rest of the steps described in Section 4 are performed. For evaluation, SimpleScalar [4] ARM was modified to implement the CCA interface and configured to match the ARM-926EJ [2]. The ARM-926EJ is a fairly simple, in-order, five-stage pipelined processor with 16K, 64-way associative in-

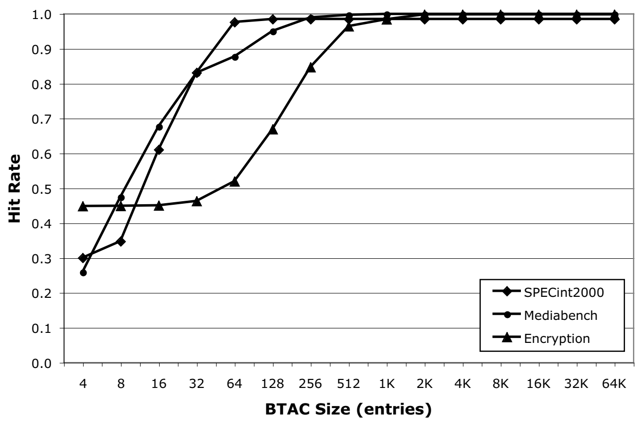


Figure 7: BTAC hit rate with various entry sizes

struction and data caches.

For our experiments, we evaluated a set of embedded and general-purpose benchmarks consisting of five encryption related applications (Blowfish, MD5, RC4, Rijndael, and SHA), and a subset of the MediaBench [17] and SPECint2000 applications. The range of our application set was limited by the current capabilities of the ARM port of the Trimaran compiler suite.

BTAC Size Study: Before evaluating the effectiveness of the CCA, we investigated several possible configurations for the BTAC, which holds the branch addresses and live-in information for the CCA subsystem. Figure 7 shows the BTAC hit rate given several different BTAC sizes. The three lines indicate the average hit rates of the BTAC for the encryption, MediaBench, and SPECint2000 applications. Interestingly, even with only 4 entries, the BTAC was able to capture a fairly large number of the marked subgraphs. For example, in the encryption domain, 45% of the subgraphs were captured. In the remaining experiments, we used to use a 512 entry, four-way associative BTAC, which achieved a hit rate average of 98.5% across all benchmarks.

Performance Study: Figure 8 shows the relative speedups that were achieved for code compiled using both basic blocks and superblocks [14]. For each benchmark, three bars are shown. The first bar is the speedup of basic block code with a CCA relative to basic block code compiled without CCA subgraphs. Both of the next two bars are superblock code with a CCA relative to superblock code compiled without CCA subgraphs. The first of the two superblock bars is for code without code motion applied, which limits the subgraphs by not allowing them to cross branch boundaries. The second superblock bar was generated by allowing the compiler to perform code motion.

All of the results in Figure 8 used the general purpose CCA designed in our previous work [7] and shown in Figure 3. Synthesis results showed that this CCA used 0.61 mm^2 of die area, and gave average speedups for the basic block code of 1.60 for SPECint2000, 1.91 for MediaBench, and 2.79 for encryption applications. The encryption applications showed the most improvement because they tend to have the largest amount of computation between memory accesses, thereby creating larger subgraphs to map onto the CCA. The results show that substantial performance gains across a wide range of applications are realized with a relatively inexpensive compute accelerator that is tightly integrated into a processor. The CCA provides a more efficient hardware substrate to execute the subgraphs, which translates into performance gain.

One trend to note in this graph is that in many cases, using superblock code had a smaller relative speedup than basic block code. Intuitively, superblock code should result in the identification of larger patterns, which should directly translate into improved performance over the basic block code. However, register pressure is an important performance issue in the ARM processor. Forming superblocks caused an increased size in register live ranges. This increase in live range size dramatically affected the amount of register spill code which the compiler was unable to optimize using the CCA.

Applying the code motion techniques discussed in 4 to the superblock code resulted in improved performance in most cases since adding the code motion optimization allowed the compiler to find patterns which cross branch boundaries. In some cases, such as cjpeg and g721encode, the performance improvement was as much as 50%, while a few other cases suffered slight performance degradation. This performance degradation is a result of code motion enlarging register live ranges as operations are pushed down below branches and new code is inserted in the target blocks. Again this increases register pressure and may lead to increased spill code.

Custom CCA Designs: Though a general purpose CCA design provides impressive performance gains across a diverse set of applications, tailoring a CCA to either a single application or a domain of applications can yield a more area-efficient design. In order to explore the design of application and domain specific CCAs, the compilation process was augmented so that when subgraphs are identified, the operations which comprise the subgraph and their profile weights are passed to a scheduler. The scheduler then incrementally builds a reservation table for each subgraph. After all subgraphs in the application have been identified, the scheduler then builds the application-specific CCA structure as the union of all of the necessary reservations for each subgraph meeting a minimal profile weight requirement. Lastly, domain specific CCA structures are built as the union of all application specific CCAs synthesized for a particular domain. This approach is not intended to produce optimal CCAs, but rather illustrate the flexibility of the proposed architectural framework to support a wide variety of CCA designs.

Figure 9 demonstrates the structure of a set of automatically generated application and domain specific CCAs. The top row of Figure 9 consists of one application specific CCA designed for an application in each of the presented domains, encryption, audio, and SPECint, respectively, while the bottom row consists of the set of domain specific CCAs.

Table 1 presents an analysis of the design costs for each of the CCAs shown in Figure 9. The table includes the number of control bits necessary to configure the CCA, the delay through the CCA, and the area of the CCA. Each of these designs was synthesized with Synopsys design tools using a 130nm Artisan library. In order to provide insight into the cost of adding a CCA to an actual ARM core, we note that the actual area of an ARM-926EJ is 5.0 mm^2. Also important to note is that the design for the general purpose CCA from [7] was hand-tuned to minimize the number of levels including adders in the CCA thus significantly reducing de-

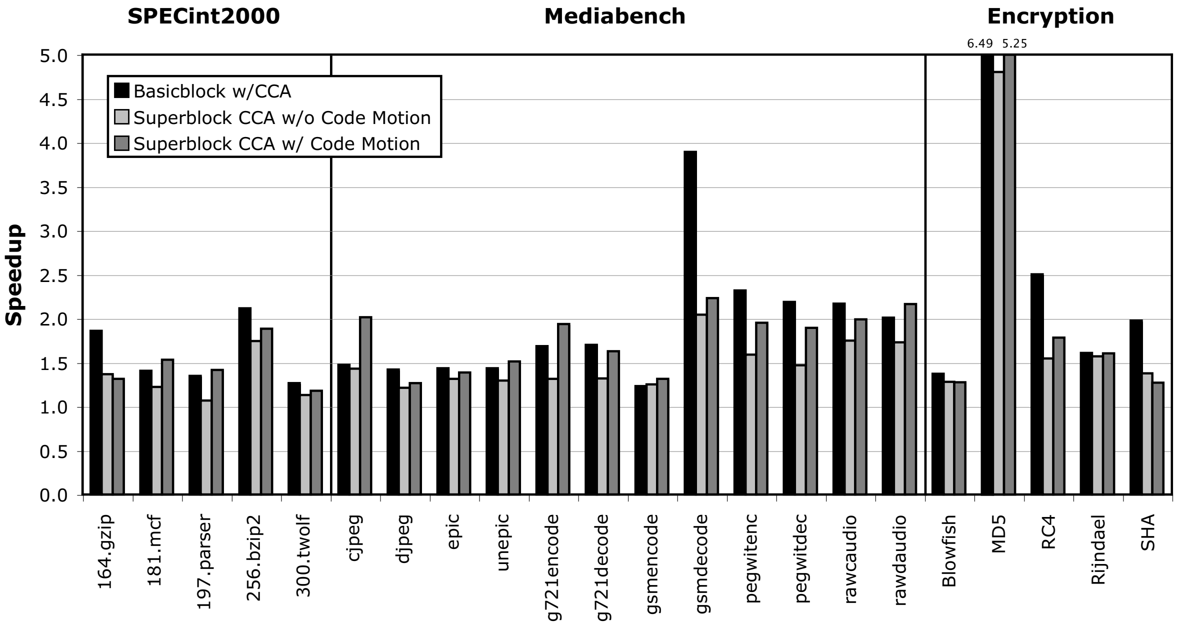


Figure 8: Speedup of basic block and superblock code when executing with a general purpose CCA

Description	Design	Control	Delay	Cell area
Application specific CCA for RC4	Figure 9(a)	73 bits	4.10 ns	0.25 mm^2
Application specific CCA for gsmdecode	Figure 9(b)	84 bits	6.04 ns	0.33 mm^2
Application specific CCA for 181.mcf	Figure 9(c)	55 bits	5.68 ns	0.26 mm^2
Domain specific CCA for encryption	Figure 9(d)	181 bits	5.69 ns	0.45 mm^2
Domain specific CCA for audio	Figure 9(e)	140 bits	5.86 ns	0.46 mm^2
Domain specific CCA for SPECint	Figure 9(f)	171 bits	6.05 ns	0.56 mm^2
General purpose CCA from [7]	Figure 3	172 bits	3.19 ns	0.61 mm^2

Table 1: Synthesis results for various CCA designs

lay through the CCA. A more intelligent automated design process for our application and domain specific CCAs would likely provide improvements in terms of both area and delay.

Figure 10 demonstrates the performance improvements offered by the designs shown in Figure 9. In this graph, the first bar indicates the performance of the general purpose CCA relative to the baseline processor with no CCA. The second bar demonstrates the speedup achieved by using the domain specific CCA designed for the domain that the application belongs to, assuming a 1-cycle delay through the CCA. The third bar demonstrates the performance of the same CCA as the second, but assumes a 2-cycle delay through the CCA. The fourth bar shows the speedup of using the application specific design shown in Figure 9 for each application in the same domain. This means that for applications within the SPECint domain, all application specific speedup is calculated using the CCA designed for 181.mcf, for the audio domain using the CCA designed for gsmdecode, and for the encryption domain using the CCA designed for rc4. The decision to use the application specific design for a variety of different benchmarks was to show the applicability of these designs across a set similar benchmarks. The last bar utilizes the same CCA structure as the fourth, but assumes a 2-cycle delay through the CCA.

From Figure 10, it is clear that a domain specific CCA design can closely match the performance of the general purpose design at lower cost, provided that it can fit into the 1-cycle delay constraint. Further, the application specific CCA designs tend to closely track the performance of their respective domain specific designs while still proving beneficial to a variety of other applications within their domain at nearly half the area overhead. It is important to note that the domain specific designs tend to provide marginal performance gains over their application specific counter parts due to their ability to catch the few subgraphs that had been pruned from the application specific CCA design.

6. CONCLUSIONS

In this work, we present the design and implementation of a flexible architectural framework for supporting *transparent instruction set customization* using *configurable compute accelerators*. The use of this framework reduces both system design and verification costs. A general purpose core implementing the pre-defined CCA interface need only be designed and verified once. The core may then be augmented with several different styles of compute accelerators offering a wide range of systems with performance characteristics tailored to an application or domain of applications. In ad-

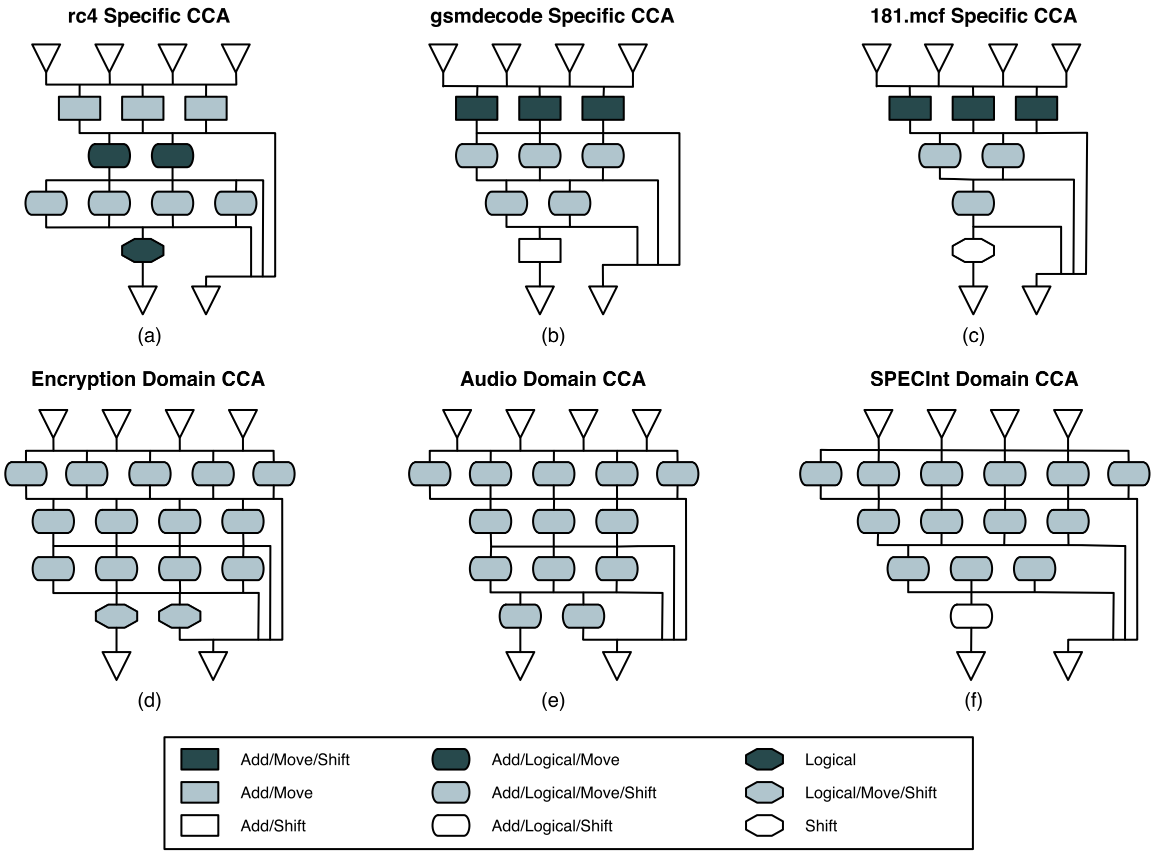


Figure 9: Application specific and domain specific CCA design results

dition to the architecture framework, we also demonstrate the compilation process used to target an application toward a particular CCA architecture.

Synthesis results demonstrate the feasibility of the proposed architecture framework in terms of meeting the timing and area constraints of common embedded processors. Further, experimental results demonstrate average performance gains of 2.21x for domain specific CCA designs, with modest cost overhead beyond the original processor design. The range of applicability of these designs may be restricted or expanded in order to both meet area constraints and satisfy performance goals for a specified range of applications. The proposed architectural framework provides system designers with a low-cost solution for designing a wide variety of high-performance systems by augmenting a single core with multiple implementations of the CCA subsystem.

7. ACKNOWLEDGMENTS

Assistance in synthesizing the CCA was provided by Hyunchul Park. Additional thanks go to Sami Yehia 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] 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] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[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] P. Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 International Conference on on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.

[7] 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.

[8] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages

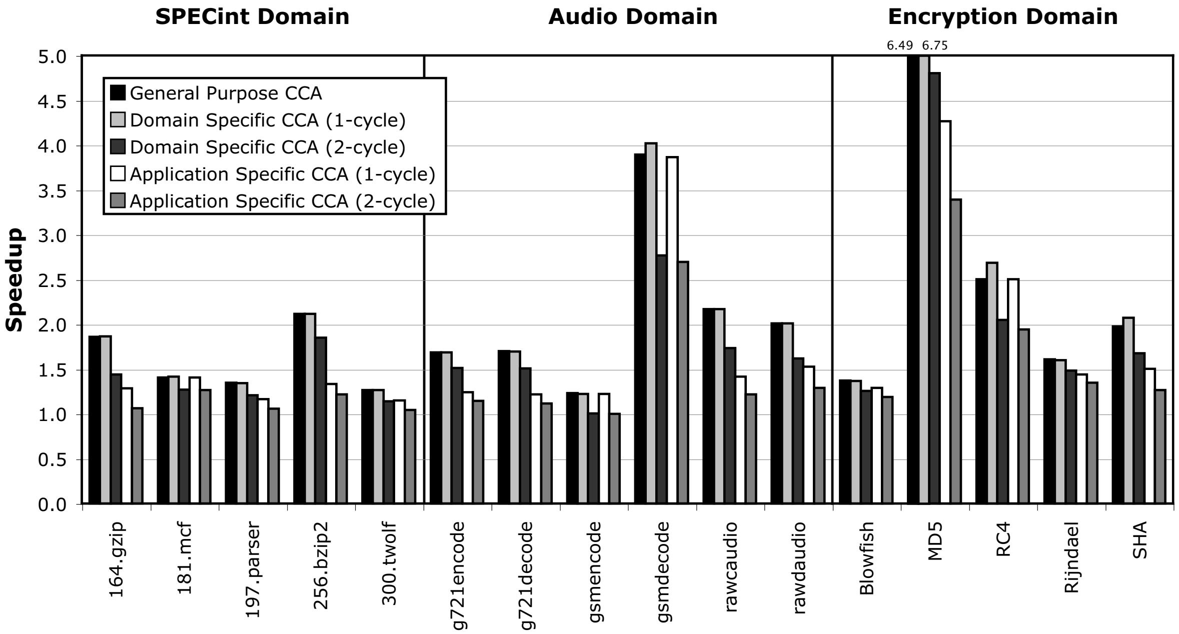


Figure 10: Application specific and domain specific speedup. For the SPECint domain, application specific speedups are generated using the CCA designed for 181.mcf, for the audio domain using the design for gsmdecode, and for encrypt domain using the design for RC4.

129–140, Dec. 2003.

[9] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 362–373, 2003.

[10] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(9):478–490, July 1981.

[11] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[12] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. of the 2003 International Conference on on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–147, 2003.

[13] I. Huang. *Co-Synthesis of Instruction Sets and Microarchitectures.* PhD thesis, University of Southern California, 1994.

[14] W. Hwu et al. The Superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993.

[15] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proc. of the 5th International Symposium on on High-Performance Computer Architecture*, pages 125–133, 1999.

[16] K. Kunchithapadam and J. R. Larus. Using lightweight procedures to improve instruction cache performance. Technical Report CS-TR-1999-1390, Jan. 1999.

[17] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[18] S. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors.* PhD thesis, Massachussetts Institute of Technology, 1996.

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

[20] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, Boston, 1995.

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

[22] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing alu's. *IEEE Trans. Comput.*, 43(3):257–268, 1994.

[23] P. Sassone and D. S. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 7–17, Dec. 2004.

[24] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, 1996.

[25] F. Sun et al. Synthesis of custom processors based on extensible platforms. In *Proc. of the 2002 International Conference on on Computer Aided Design*, pages 641–648, Nov. 2002.

[26] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org.

[27] 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 31th Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.

[28] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proc. of the 41st Design Automation Conference*, pages 723–728, June 2004.