VEAL: Virtualized Execution Accelerator for Loops

Nathan Clark[1], Amir Hormati[2], and Scott Mahlke[2]

[1]College of Computing
Georgia Institute of Technology
ntclark@cc.gatech.edu

[2]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor
{hormati, mahlke}@umich.edu

Abstract

Performance improvement solely through transistor scaling is becoming more and more difficult, thus it is increasingly common to see domain specific accelerators used in conjunction with general purpose processors to achieve future performance goals. There is a serious drawback to accelerators, though: binary compatibility. An application compiled to utilize an accelerator cannot run on a processor without that accelerator, and applications that do not utilize an accelerator will never use it. To overcome this problem, we propose decoupling the instruction set architecture from the underlying accelerators. Computation to be accelerated is expressed using a processor's baseline instruction set, and light-weight dynamic translation maps the representation to whatever accelerators are available in the system.

In this paper, we describe the changes to a compilation framework and processor system needed to support this abstraction for an important set of accelerator designs that support innermost loops. In this analysis, we investigate the dynamic overheads associated with abstraction as well as the static/dynamic tradeoffs to improve the dynamic mapping of loop-nests. As part of the exploration, we also provide a quantitative analysis of the hardware characteristics of effective loop accelerators. We conclude that using a hybrid static-dynamic compilation approach to map computation on to loop-level accelerators is an practical way to increase computation efficiency, without the overheads associated with instruction set modification.

1 Introduction

For decades, industry has produced, and consumers have relied on, exponential performance improvements from microprocessor systems. This continual performance improvement has enabled many applications, such as real-time ray tracing, that would have been computationally infeasible only a few years ago. Despite these advances, many compelling application domains remain beyond the scope of everyday computer systems, and so the quest for more performance remains an active research goal.

The traditional method of performance improvement, through increased clock frequency, has fallen by the wayside as the increased power consumption now outweighs any performance benefits. This development has spurned a great deal of recent research in the area of multicore systems: trying to provide efficient performance improvements through increased parallelism.

Not all applications are well suited for multicore environments, though. In these situations, an increasingly popular way to provide more performance is through customized hardware. Adding application specific integrated circuits (ASICs) or application specific instruction set processors (ASIPs) to a general purpose design not only provides significant performance improvements, but also major reductions in power consumption as well. There are many examples of customized hardware being effectively used as part of a system-on-chip (SoC) in industry, for example the encryption coprocessor in Sun's UltraSPARC T2 [23].

The main drawback of this approach is that creating specialized hardware accelerators for each targeted application carries significant costs. Hardware design and verification effort, software porting, and fabrication challenges all contribute to the substantial non-recurring engineering costs associated with adding new accelerators. Purchasing accelerator designs, in the form of intellectual property (IP), is a popular option to alleviate some of the hardware design costs, but there are still significant integration costs (both hardware and software) in tying the IP accelerators into the rest of the system.

The goal of this work is to attack those costs. First, we present the design of a hardware accelerator that effectively executes a class of loop bodies for a range of applications. Many applications spend the majority of their time executing in innermost loops, and so ASICs tend to implement one or more loop bodies. By defining a single architecture to accelerate loops, the recurring costs of designing an application specific accelerator are eliminated. The goal is to cost-effectively generalize an ASIC design to make it useful for a wider range of loops, without generalizing it to the point where it begins to look like a general purpose processor.

The second step is to attack the software costs of targeting an accelerator. Software costs primarily result from re-engineering the application once the underlying hardware has changed. To avoid these costs, we develop a software abstraction that virtualizes the salient architectural features of loop accelerators (henceforth abbreviated LAs). An application that uses this abstraction is dynamically retargeted to take advantage of the accelerator if it is available in the system; however, the application will still execute correctly without any accelerator in the system. The tradeoff is to abstract away as many architecture-specific features as possible without requiring a significant overhead to dynamically retarget the application.

The resulting design is referred to as *VEAL*, or virtualized execution accelerator for loops. There are two primary contributions of this work:

- It presents the design of a novel loop accelerator architecture. Quantitative design space exploration ensures that the accelerator design is broad enough to accelerate many different applications, yet very efficient at executing the targeted style of computation.
- It describes a dynamic algorithm for mapping loops onto loop accelerators. The algorithm is analyzed to determine the runtime overheads introduced by this dynamically mapping loops, and static/dynamic tradeoffs are investigated to mitigate the overhead.

2 Overview

It is widely acknowledged that the vast majority of execution time for most applications is spent in loops. Applying this fact, along with Amdahl's Law, often leads system designers to construct hardware implementing loop bodies whenever processors are insufficient to meet performance or power consumption goals. For example, special purpose LAs for Fast Fourier Transforms and Viterbi decoding are common in many embedded SoCs.

This section begins by describing the general architecture common across LAs. Next, it gives an overview of *modulo scheduling*, a compilation technique used to schedule loops so that they effectively use the hardware resources available to them. The section concludes by introducing the issues surrounding dynamically retargeting applications to a particular LA implementation, which are fully discussed in Section 4.

2.1 Loop Accelerator Architectures

In order to determine an appropriate architecture for a broad set of loop bodies, it is first necessary to understand the general structure of LAs. The left portion of Figure 1 shows the high level structure of an LA. At the top of the accelerator portion of this figure, address generators stream data into the accelerator. The address patterns typically follow a simple, deterministic pattern (often based on the loop's induction variable(s)) that enables them to be decoupled from the computation performed on the data. When data is streamed in from the memory system, it is placed in FIFOs that are accessed by function units (FUs). Address generators can be time multiplexed to fetch multiple streams, which enables them to hide any stalls due to bursty memory behavior amongst the different streams, and reduce the number of memory ports required. Input data that is not streamed into the accelerator, such as constants or scalar inputs, are written into a register file. Typically, this register file is memory mapped and must be initialized before invoking the accelerator.

Once data is available, FUs begin processing it, reading values from the FIFOs or register file and writing results to either the output memory buffers or registers. The register file that stores results from the FUs, need not be a monolithic standard SRAM; many LAs utilize distributed SRAMs [4] or more efficient structures such as FIFOs [11] or ShiftQs [1]. Additionally, the functionality provided by the FUs is often highly customized, executing several RISC-equivalent operations (ops) back-to-back in the name of improved efficiency [28].

Once computation has completed, another set of address generators stream the results back to memory. Input and output memory streams can optionally be assumed mutually exclusive, so that the accelerator does not need to perform memory dependence analysis, although this will preclude certain types of loops. Branches within the loop body are fully predicated enabling very simple logic in the accelerator.

Generalized LAs typically provide an interface with the system in the same way that current ASICs interface with SoCs: invoking the accelerator is an atomic act, meaning there is no need to support precise exceptions in the middle of accelerator execution; an exception either waits for the accelerator to complete, or aborts the accelerator's current execution. The accelerators also operate using physical addresses, so that no address translation is needed, and the accelerator cannot cause page faults. Physical addressing is standard for ASICs, but if the operating system issues created through this decision are too onerous, then a virtual address strategy similar to the one by Cell's SPEs [12] could be used instead.

This abstract architecture encapsulates the structure of most loop-targeting ASICs [27] as well as previously proposed generalized LAs [3, 4, 20] targeting scientific and media processing applications. It should be noted that this architecture is primarily targeting these two domains, and is not meant to target *all* loops, such as linked-list traversals.

There are several reasons why this architecture is more efficient at executing loops than general purpose processors. First, the control flow in loops is very simple, removing the need for control flow speculation such as sophisticated branch predictors. Second, the repeating control sequence (instructions) used to configure the accelerator can be stored in a circular buffer, which is more efficient to access than a large instruction cache. Third, the memory accesses that are not data dependent may be independent from each other, enabling memory accesses to be decoupled from the computation. Lastly, the interconnect, FUs, and register files can be customized to fit the needs of the application or domain that is being targeted.

2.2 Utilizing Loop Accelerators

Assuming that there is an effective piece of hardware for executing loops, it is also necessary to have a capable compilation strategy to make use of the hardware. Modulo scheduling is a state-of-the-art software pipelining heuristic for scheduling loops, and provides the basis for the software techniques presented in this paper. Previous work on modulo scheduling is extensive [7, 16, 17, 19, 24, 25, 26], and the purpose of this section is only to introduce fundamental concepts.

Modulo scheduling is a method of overlapping loop iterations to achieve high throughput. The instructions are assigned to FUs so that new iterations can begin executing at a constant rate, called the *initiation interval*, or simply *II*. A software pipeline is divided into three distinct parts. The periods where the software pipeline is ramping up and ramping down are called the *prologue* and *epilogue*, respectively, and the steady state (when an iteration is starting and completing every II cycles) is called the *kernel*.

A single loop iteration can be broken down into multiple *stages* based on how many times II cycles has passed since it began executing. The different time steps in each stage are referred to as the stage *cycles*, which range from 0 to II-1. The goal of modulo scheduling heuristics is generally to make II as low as possible, so that kernel execution is reached and completed as soon as possible. A secondary goal is to make the number of stages (often abbreviated *SC* for stage count) as small as possible. To rephrase using standard pipeline terminology, lower II equates to higher iteration throughput and lower SC equates to lower iteration latency.

While modulo scheduling has proven to be effective, there are some limitations with the process. One limitation is that loops with function calls cannot be modulo scheduled. This problem can mitigated through intelligent function inlining, and is not a major drawback. A more important limitation is that while-loops and loops with side exits require special hardware support, such as speculative memory accesses [21, 24]. Although it is feasible to support while-loops and loops with side exits, we chose to preclude them from this study, to minimize the architectural impact outside the accelerator itself.

Figure 2 demonstrates the implication of this decision. Each bar in this figure represents the entire execution time for a given benchmark from MediaBench or SPEC.[1] The black bars on the bottom are the fraction of time spent executing in modulo schedulable loops. The bars labeled "Speculation Support" refer to the time spent in while-loops that would be modulo schedulable, provided the appropriate hardware support existed. Bars labeled "Subroutine" are loops with function calls that could not be inlined (e.g., calls into the math library that were not visible to the compiler).

Media processing and floating point applications (the left portion of Figure 2) tend to spend the vast majority of their execution time in modulo schedulable loops. Lack of support for loops requiring speculation will limit the utility of the LA for

[1]A variety SPECint and SPECfp from the 92, 95, and 2000 suites were chosen for this study. We omitted benchmarks that could not be compiled with Trimaran [30].

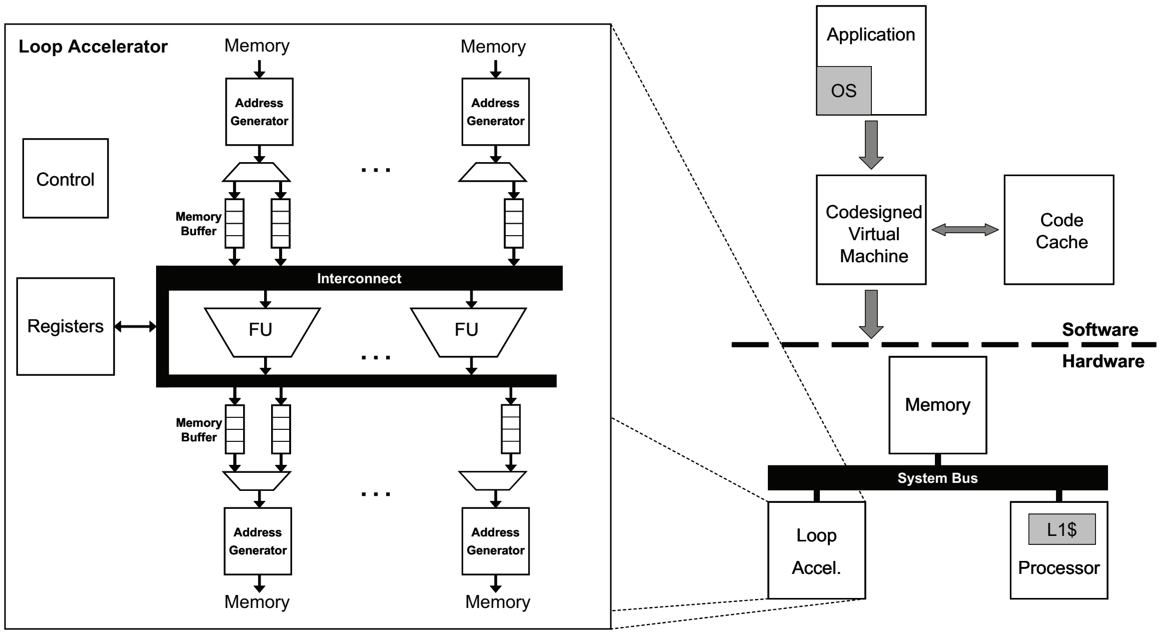


Figure 1. Architecture template for loop accelerators (left) and VEAL system organization (right).

some applications (e.g., the applications on the right portion of Figure 2); however, modulo schedulable loops clearly represent an important class of computation worthy of hardware acceleration.

2.3 Dynamic Retargeting

Using specialized hardware to execute loops has many performance and power benefits, but hardware and software design costs prevent widespread deployment in many cases. Designing one architecture for a broad set of loops tackles the hardware design costs. However, the software design costs remain a difficult problem.

Software costs arise from the fact that the control to invoke a LA is statically encoded in the binary. An application that utilizes an accelerator typically has no forward or backward compatibility. This means that whenever the underlying hardware platform changes, the application must be re-engineered. While some markets, notably embedded systems, have traditionally been willing to bear this overhead, high software cost prevents the use of accelerators in many markets that are not large enough to justify the effort, and this inhibits innovation.

The standard technique for proving architecturally independent access to customized hardware is through a library interface. However, this is not flexible enough, because the library interface often assumes a specific execution model in the underlying hardware that prohibits future hardware innovation. Additionally, libraries impede optimization across interface boundaries, yielding highly suboptimal code (e.g., much of the computation in sin(x) and cos(x) is identical, but if an application had consecutive calls to these functions, it would not take advantage of the redundant work). Architects need to provide more flexible interfaces to customized hardware.

The method we propose to avoid the software engineering costs is to virtualize the accelerator interface. That is, we will analyze the steps used during compilation to map applications onto LAs, and perform as many of them dynamically as possible as part of a *co-designed virtual machine* (shown in the right portion of Figure 1). The loops to be accelerated are encoded using the baseline instruction set of a general purpose processor. At runtime, a co-designed virtual machine (VM) monitors execution of the application and dynamically generates control for the accelerator when appropriate. This dynamic software layer enables the binary flexibility, unchaining the binary from any one specific accelerator architecture, and allows the binary to run on systems without any accelerator at all.

The challenge in virtualization is to determine the appropriate static/dynamic tradeoffs to make in the binary. High quality modulo scheduling heuristics can be sophisticated, taking too long to perform dynamically. If the translation takes too long, it can completely erode all the efficiency benefits from using the accelerator in the first place. At the other end of the spectrum, performing the mapping entirely statically ties the binary to a single accelerator implementation, which has significant nonrecurring engineering costs if the underlying hardware changes.

The remainder of this paper is organized as follows: Section 3 performs a design space exploration for a generalized LA for a range of media and floating point applications. This design provides the basis for our work on virtualization, which is covered in Section 4. Section 4 walks through the details of one particular modulo scheduling heuristic and analyzes the tradeoffs involved in performing each step statically versus dynamically in a co-designed VM.

3 Generalized Loop Accelerator

To mitigate the costs of customized hardware, it often makes sense to extend the programmability of ASICs, making them more useful across a broader set of applications. The goal of this section is to do just that: design an architecture that effectively supports the set of modulo schedulable loops from the MediaBench and SPECFP applications on the left portion of Figure 2. Designing this architecture has two purposes. First, we provide a quantitative analysis of the tradeoffs involved with adding each execution resource to the accelerator. Previous work [3, 20] designing generalized LAs presented designs without this analysis. Second, this design helps gauge the static/dynamic tradeoffs in modulo scheduling to target an LA. It would stand to reason that the time needed to modulo schedule a loop is correlated with the number and type of resources in the target machine, and so a representative architecture is necessary to accurately measure translation overheads.

The LA architecture template shown in Figure 1 will serve as the basis for our generalized design. Customizing the tem-

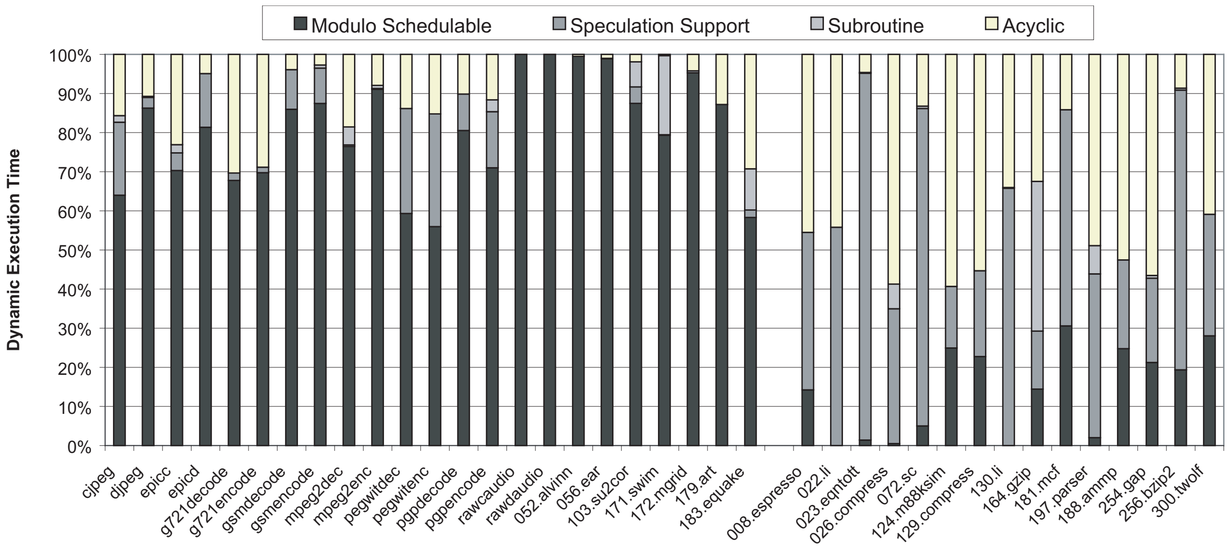


Figure 2. Percent of execution time spent in various types of code. "Speculation Support" refers to while-loops and loops with side exits, "Subroutine" refers to loops that have a non-inlinable function call, and "Acyclic" refers to code not known to be in a loop.

plate for the targeted application set now requires identifying how many resources of each type these applications require. To determine this, we modified the Trimaran toolset [30] to compile for and simulate a processor with attached LA. The accelerator connects to the processor through a system bus using a memory mapped interface. All speedups reported in this paper are for entire applications, not just loop bodies, and include synchronization overheads from copying results to and from the accelerator over a 10 cycle system bus.

3.1 Design Space Exploration

The baseline architecture in our design space exploration assumes a hypothetical LA with infinite resources. That is, loops are modulo scheduled onto a machine with unlimited registers, FUs, memory ports, etc. Architectural parameters were then individually varied to determine what fraction of the infinite-resources speedup was attainable using finite resources (the actual speedup numbers are presented in Section 4). The modulo scheduling heuristic used to target each application to the accelerators is discussed in detail in Section 4.1. As previously mentioned, only the benchmarks on the left portion of Figure 2 were used in this analysis.

Figures 3(a) and 3(b) show the results of the design space exploration for FUs and register requirements. The x-axis in these graphs represents the number of resources available in the system and the y-axis is the fraction of infinite-resource speedup attained. For example, the gray line in Figure 3(b) shows that when there is only one floating-point register, the average speedup across the targeted application suite is 60% of what is attainable with infinite floating-point registers.

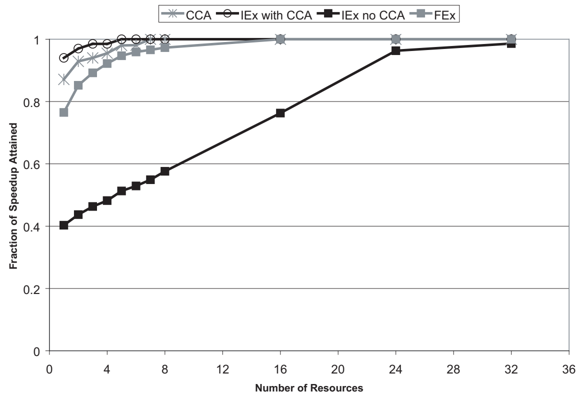
Figure 3(a) explores the FUs available in the accelerator, where IEx and FEx represent integer and double-precision floating-point units, respectively. One interesting result from this experiment was that very few floating-point units (FEx in the left graph) were needed to attain a significant amount of speedup in the application set. This is partially due to the significant number of integer-only applications in our target suite, but the long latency of floating-point operations also contributes to this result. If a floating-point unit is fully pipelined (which was assumed), modulo scheduling does a very good job utilizing the unit every possible cycle.

One surprising result from Figure 3(a) is that the point of diminishing returns for integer units is very high, on the order of 24 units. Due to this result, we chose to experiment with another type of FU, a CCA [5]. The CCA is a combinational structure specifically designed to efficiently implement the most common types of integer computations. It supports 4 inputs, 2 outputs, and can execute as many as 15 standard RISC ops atomically in 2 clock cycles. The 15 RISC ops are organized into 4 rows, where the first and third row can execute simple arithmetic (add, subtract, comparison) and bitwise logical ops, and the second and fourth rows execute only bitwise ops. The primary benefits of the CCA result because it executes larger pieces of computation as a group, reducing storage and interconnect requirements, as well as squeezing more work out of each clock cycle. The top line in Figure 3(a) shows that when one CCA is added to the LA, the required number of integer units drops dramatically. Some integer units are still needed to support multiplication and shifts, which are not handled by the CCA.

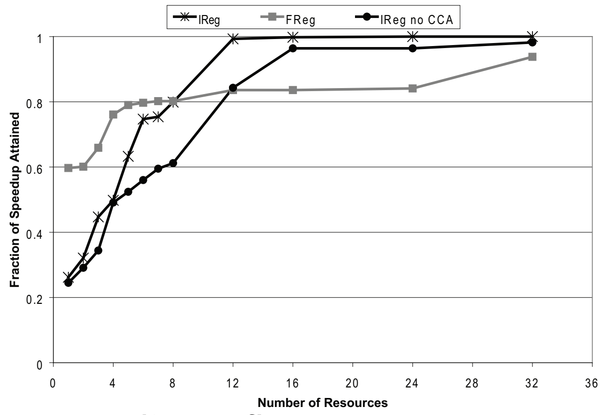
Figure 3(b) shows the required number of registers needed to store live-ins, live-outs, constants, and temporary values for the loop. Note that registers are not needed to store values that are read from or written into memory FIFOs nor are they needed for values that are read directly off the interconnection network (i.e., values computed the previous cycle). When there are not enough registers to support a loop, it is simply run on the baseline processor. Overall, few registers are needed to support the majority of important loops. As would be expected, adding a CCA to the system reduces the register requirements, since fewer temporaries are needed to communicate between separate units.

Similar to Figures 3(a) and 3(b), Figure 4(a) shows the the fraction of infinite-resource speedup attained when varying the number of load/store streams supported in the accelerator.[2] In this analysis, we define a stream as a unique reference pattern, i.e., a base address and a linear function that modifies that address each loop iteration. As would be expected, loads are more important than stores. Surprisingly, many loops can be

[2]Note, the number of streams is not equal to the number of memory ports. Many streams will share a time-multiplexed memory port in the design.

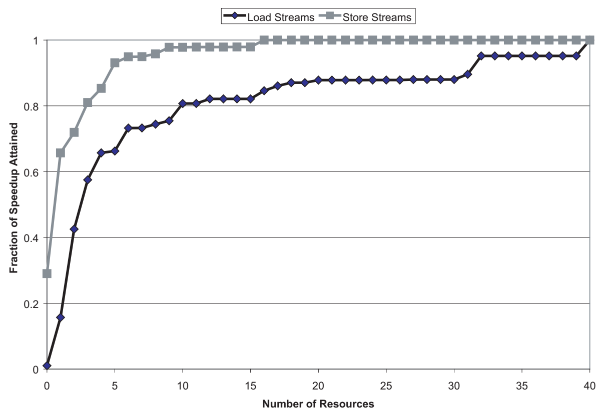


(a) Function unit requirements

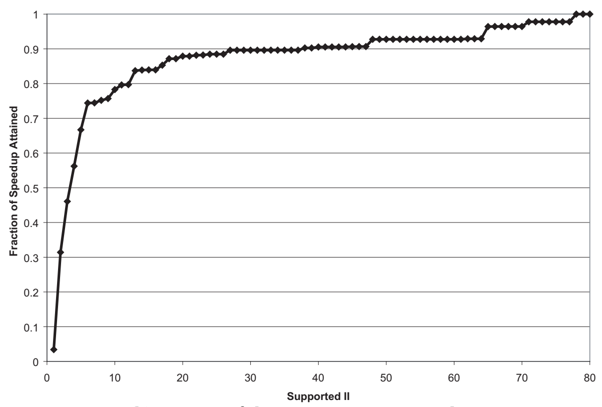


(b) Register file resource requirements

Figure 3. Execution resource needs. Each line is the fraction of infinite-resource loop accelerator speedup attained when varying the number of function units



(a) Memory stream resource requirements



(b) Impact of the maximum supported II

Figure 4. Resource requirements for a generalized loop accelerator

supported without any address generators streaming data out to memory; these loops only have scalar outputs, which are read directly from the memory mapped register file upon loop completion.

Another surprising result from the memory stream analysis is that a very large number of memory streams were needed to support several important loops in the examined benchmarks. For comparison purposes, previously proposed general LAs only supported 3 load/1 store [3] stream or 6 total load/store streams [20] per loop. Supporting fewer memory streams is desirable, since it requires less hardware (e.g., storage for base addresses and access patterns).

Empirically speaking, the loops that required a large number of memory streams tended to be very large. These large loops typically resulted from aggressive function inlining used by the compiler to improve loop accelerator utilization in the target applications. One potential way to reduce the hardware overhead of supporting these large loops is to time-multiplex the address generators. Large loops tend to have larger IIs, giving the address generators time to process several different streams. Another potential solution is to break the large loops up into smaller loops using a technique such as loop fissioning. This would reduce the required number of streams for each individual loop but increase memory traffic, as dividing the loop up typically creates communication streams between the smaller loops.

The graph in Figure 4(b) shows the maximum supported II by the LA (i.e., loops that cannot be scheduled at the maximum II will not be accelerated). This essentially demonstrates the longest recurrence paths in the studied loops since recurrence-free loops can always be scheduled at an II of 1 with infinite resources.

Maximum supported II is an important consideration in the loop accelerator design, because the size of the loop control is directly proportional to the maximum supported II. Since the kernel repeats every II cycles, each FU needs to be able to execute II different instructions, and thus maximum supported II determines the size of the control structure. As with the memory stream limitation, if a particular loop is too large to be supported by an II, often times proactive loop fissioning enables the loop to utilize an accelerator.

3.2 Loop Accelerator Design

Using the analysis in this section as a guide, we propose a generalized LA design consisting of 1 CCA, 2 integer units, 2 double-precision floating-point units, 16 floating-point and integer registers, 16 load memory streams (time-multiplexed among 4 address generators), 8 store memory streams (time-multiplexed among 2 address generators), and a maximum II of 16. This is sufficient for attaining 83% of the speedup possible using a hypothetical loop accelerator with infinite resources. To estimate the die area impact of this design, estimates of the constituent parts were collected using Cadence tools and a IBM 90 nm standard cell library. We project the design would consume 3.8 mm^2 of die area, the majority of that (2.38 mm^2) being consumed by the two double-precision floating point units.

To put these numbers in perspective, the ARM 11 processor (a single-issue embedded processor with 8 stage pipeline, 16K L1 caches, 128K L2 cache, and no FPU) consumes 4.34 mm^2 in a similar process. An ARM Cortex A8 processor (dual-issue, 13-stage pipeline, 32K L1 caches, 256K L2 cache) would consume roughly 10.2 mm^2, meaning that the loop accelerator could be added to an embedded system for less than the cost of a second simple core, or the cost of increasing the cache and issue width.

The design space exploration presented here has omitted two major portions of the data path: the register file structure and interconnect customizations that often occur in customized hardware accelerators. The primary reason for this omission is that there are currently few modulo scheduling algorithms that take these customizations into consideration. Without software support to analyze the costs of architectural customization (in terms of reduced performance), it is difficult to make intelligent design decisions, and this exploration is left for future work.

4 Virtualizing the Accelerator

The LA architecture is very effective at executing the modulo schedulable loops from the wide range of applications studied. However, the tradeoffs made in that design will not fit all situations. When this is the case, a new accelerator must be designed for the system, which creates a burden on the application developer. Traditionally, control used to invoke an accelerator is statically placed in the binary, meaning the application will have to be re-engineered to function on a different hardware platform. This software porting cost often prevents hardware innovation in situations where it otherwise provides significant benefits.

The way to eliminate the software cost is to generate the control for the accelerator dynamically, only after the application knows what accelerators are available in the system. This type of system has previously been termed a co-designed VM [29], because software dynamic translation is designed in conjunction with new hardware features.

Dynamically generating control for the LA relies on the assumption that the cost of performing the translation is low; otherwise, the translation cost would outweigh any benefits provided by the custom hardware. Thus, the key to virtualization of custom hardware is analyzing the algorithms used to generate control, performing the time consuming parts statically, and encoding them in the binary in a way that is binary compatible with other systems.

Towards this end, this section first walks through an example demonstrating the translation process. Later, this section explores the implications of performing each translation step statically versus dynamically to develop an appropriate machine-independent interface.

4.1 Loop Accelerator Translation

Identifying and Transforming Hot Loops: Many steps are necessary to retarget an application to leverage loop accelerators; these are illustrated in Figure 5. The initial step is simply to identify loops within the program that can potentially be mapped onto the accelerator. Loop identification (i.e., finding strongly connected components of a control flow graph) is a simple linear time problem, and common in nearly all compilers.

Once loops are identified, they must be checked to ensure that the LA provides sufficient features to support the loop. For example, the loop may require more load streams than the accelerator can support, or have function calls within the loop body. Generally speaking, there are proactive compilation methods to make these ostensibly unsupported loops execute on the accelerator. For example, a loop with too many load streams can potentially be broken into one or more smaller loops by loop fissioning; function calls can often be inlined to remove the control flow exit from the middle of the loop, as well.

Separating Control and Memory Streams: After transforming the loop to fit the accelerator, data dependence information is used to identify the control and address calculations. These calculations are then mapped onto the special hardware supporting address generation and accelerator control. The example loop in Figure 5 has op 15 as the loop-back branch. Following the backward slice of dependence edges from that branch delineates a simple control pattern where op 13 increments an induction variable and op 14 compares it to a terminating condition. Likewise, loads and stores (ops 2 and 12) are followed to identify their address computation patterns (ops 1 and 11 in this example). If the control and address patterns are more complicated than supported by the accelerator, then translation terminates at this point.

Mapping address computations to the hardware can potentially be more complicated than control depending on the level of support provided in the accelerator. For example, if the address calculation units do not have memory ordering hardware (i.e., there is no support for the equivalent of a load-store queue), then the mapping algorithm must provide guarantees that decoupling the load/store streams will never cause dependency violations. Compiler-based memory disambiguation has proven very challenging over the years, and thus we assume hardware support for memory ordering exists.

CCA Mapping: The next step in compiling for the accelerator is to attempt to collapse multiple RISC instructions into a single CCA instruction (if a CCA is present in the system). The CCA is designed to efficiently execute larger pieces of integer computation, thus moving computation to this resource improves the loop schedule. Optimally utilizing the CCA is an NP-complete problem [13], so this work uses a greedy algorithm to keep runtime overheads low.

CCA mapping begins by selecting a seed node in the dataflow graph. In the example loop in Figure 5, seed ops are examined in numerical order. Op 5 is selected as the first seed, since the targeted CCA from Section 3 does not support shifts or multiplies needed to execute ops 3 and 4. This seed is then recursively grown along its dataflow edges to extend the subgraph to include ops 8 and 6, which are supported by the CCA. Once the subgraph cannot be grown further, those three ops are replaced with a new CCA instruction, op 16, and the process begins with a new seed, op 7. Ops 7 and 10 could legally be combined; however, doing so would lengthen one of the recurrence cycles, which may increase II.

After identifying subgraphs for execution on the CCA, modulo scheduling begins. Modulo scheduling is a *class* of software pipelining heuristics, and it was necessary to choose one heuristic as the foundation of this study. We chose the Swing modulo scheduling algorithm [19] because previous work [7] demonstrated that it produces high quality schedules and is significantly faster than other modulo scheduling algorithms, particularly when the machine has a large number of resources. Speed makes it well suited to a dynamic environment where translation overheads are important.

Minimum II Calculation: The first step in modulo scheduling algorithms is to compute the minimum II, which is a function of both the recurrences in the loop and the resources available in the accelerator. Consider the example loop in Figure 5 again. This loop has two recurrences, ops 3-16-9 and ops 4-7, which are both 4 cycles long. Because the longest recurrence is 4 cycles long, the II must be at least 4, since it is impossible to start future iterations before the recurrence completes execution. Resources may also affect the minimum II of

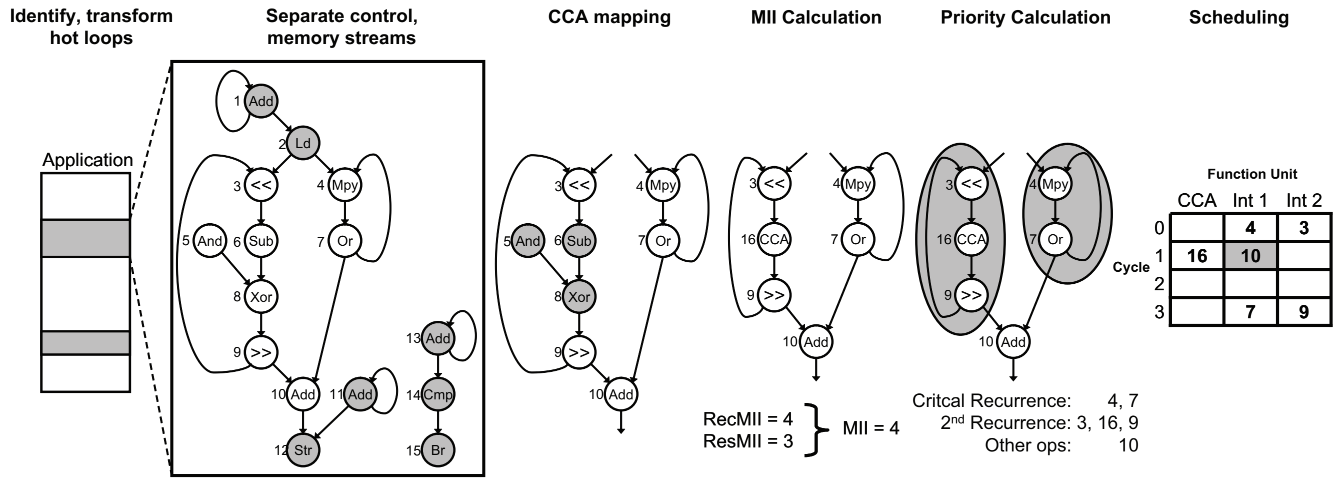


Figure 5. Scheduling an example loop body. Assume multiplies take 3 cycles, the CCA takes 2 cycles, and all other ops take 1 cycle.

loops. Using Figure 5 as an example, since there are 5 integer instructions in the loop (3, 4, 7, 9, and 10) and 2 integer units, II must be at least $\lceil \frac{5}{2} \rceil$, or 3, because an iterations worth of computation must be issued every II cycles. The minimum II for a loop is the maximum of the recurrence and resource constrained IIs (abbreviated RecMII and ResMII), or 4 in this case. A more thorough discussion of algorithms to compute II is covered in [24].

Priority Calculation: Now that II is known, ops are prioritized to determine the order in which to schedule them. Simplifying a bit, the priority function used in Swing modulo scheduling tries to schedule the most critical recurrence in a loop first, moving through less critical recurrences, and then finally to ops that do not appear on any recurrence paths. The intuition behind this is that scheduling the recurrences is a more constrained problem since the ops have a minimum and maximum schedule time. Failing to schedule a recurrence at a target II will create impossible schedules, forcing the scheduler to increase II (lowering performance) in order to translate the loop. Using Figure 5 as an example again, the modulo scheduling priority will try to schedule the most critical recurrence (ties broken arbitrarily) 4-7, followed by the next most critical recurrence 3-16-9, followed by the remaining acyclic ops.

Scheduling: Once the ops are prioritized, a modulo reservation table (shown at the right of Figure 5) is constructed to store the scheduling results. The table has II rows and a column for each FU. Ops are placed in the table using a slightly modified list scheduling algorithm. Initially, the reservation table is empty and the scheduler places the highest priority op, 4, in a schedule slot. Since op 4 requires an integer unit, and none are being used, it is placed on Int 1 at time 0. Next, op 7 is placed. Since op 7 depends on the result of op 4, a 3 cycle op, the earliest op 7 can execute is time 3, and so it is greedily placed on the first available integer unit in that cycle. The process repeats for the second recurrence 3-16-9. Finally, op 10 depends on results from ops 7 and 9, so the earliest it can be scheduled is at time 4. Time 4 is cycle 0 in the modulo schedule, though (recall, $cycle = schedule\ time \mod II$), and both the integer units are full that cycle. The scheduler increments the schedule time, and op 10 is finally placed at time 5 (cycle 1). Op 10 is colored gray in the figure to represent that it is scheduled at a different stage in the modulo schedule than the other ops. Once all the ops are placed, they represent all the control signals needed to configure the LA's datapath. Full details on how the prioritization and scheduling algorithms work can be found in [19].

Register Assignment: After a loop schedule is generated, a postpass maps operands from the loop representation in baseline assembly code to the register files/memory buffers in the LA. If there are not enough registers to support the translated loop, translation aborts, and the loop is executed on the baseline processor. In addition to operand mapping, a translator must also generate load/store instructions to move scalar inputs/outputs between the LA and the scalar processor. Control data representing the loop schedule is transferred to the loop accelerator through a memory mapped interface.

4.2 Dynamic Compilation Considerations

Now that the process of mapping loops onto the LA is clear, we must investigate the implications of performing this process in a co-designed VM. The role of the VM is to provide architectural independence for the application binary. From a high level, the VM operates by observing an application's execution and dynamically optimizing portions that benefit from acceleration. Optimized control is then placed in a software managed code cache, and the original code is modified to send a code cache pointer to the LA, starting accelerator execution. The complexities of this approach (e.g., handling I/O, supporting precise exceptions, code cache management, etc.) have been well covered in previous co-designed VM work [29], and it is not the purpose of this paper to rehash those details.

Given this framework for providing architectural independence, we shall explore the ramifications of performing each of the accelerator translation steps statically versus dynamically. Performing the translation entirely dynamically is desirable because it would provide complete architectural independence of the application binary; even legacy binaries could utilize the accelerator. However, this decision would seriously degrade the benefits provided by the custom hardware. One reason for this is the overhead needed to translate the loops.

Figure 6 demonstrates the importance of driving the translation overhead as low as possible. This graph shows the average speedup across benchmarks when varying the translation cost per loop and targeting the LA proposed in Section 3 (additional details on the experimental setup appear later in this section). The various lines reflect how frequently the translation penalty must be paid. For example, the top line assumes that each modulo schedulable loop need only be translated once during benchmark execution, and the bottom line assumes each

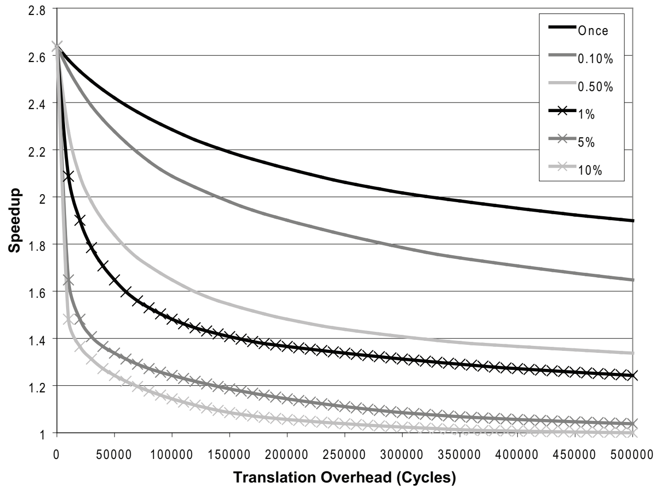


Figure 6. Speedup attained when varying the translation overhead penalty. Each line represents how frequently the penalty must be paid.

loop must be translated 10% of the times it is invoked, due to eviction from a code cache.

This figure shows that driving the translation cost very low has significant benefits. For example, if the overhead was 100,000 cycles per loop and the miss rate was 1%, lowering the overhead to 20,000 cycles increases the speedup provided by the LA from 1.47 up to 1.92. An alternate way to view Figure 6 is that it stresses the importance of providing enough space in the code cache so that loops do not need to be repeatedly translated.

Some have suggested that translation overhead is becoming less important with the proliferation of multicore processors, since one processor can run the application in parallel with the translation. While this will lessen the translation impact to some degree, it is not difficult to imagine situations where translation latency is still critically important, such as workloads with many short running tasks, or systems with frequent context switching [15].

Now that it is established that translation overhead is a critical concern, we will look at the implication of performing each step of LA translation dynamically.

Loop Identification and Transformation: Identification of loops is feasible at runtime. Several dynamic binary optimization systems already perform loop identification, as it a natural extension to region formation [2, 8, 9]. Performing this dynamically does have several drawbacks, however. As noted in [2], high quality loop transformations are much too complicated to perform in a time-constrained environment. This prevents important optimizations, such as inlining and loop fission, to help fit loops onto a targeted accelerator.

Performing loop transformations improves the utilization of the LA, but the downside is that the application will only map to accelerators that provide a superset of the capabilities of the accelerator targeted in the static compilation stage. For example, if a loop was statically fissioned so that it only used 4 load streams, then that application would not be able to use a future accelerator with only 3 load streams unless the application was statically recompiled. Because it is unlikely that the number of features would decrease as systems evolve, and complex loop transformations are important to accelerator utilization, we advocate performing loop transformations statically.

Figure 7 shows the importance of these loop transformations. Each bar in this graph shows the fraction of speedup attained by binaries without loop transforms (i.e., compiled normally) compared to binaries compiled with loop transformations (aggressive inlining, aggressive predication, and reduced unrolling) when targeting the LA from Section 3. For example, the 0 fraction shown by many benchmarks in this figure means that the runtime system was not able to retarget any of the important loops in the application without proactive help from the compiler. On average, not performing loop transformations reduced speedup attained by the accelerator by 75%, demonstrating that it is critically important to perform the transformations. The loop transformations do not functionally change the code, and no special encoding is needed to represent them in the baseline instruction set of the processor. Loop detection remains dynamic, as it is a low-overhead process to perform in the VM.

Modulo Scheduling: In order to gauge the overheads associated with dynamically modulo scheduling loops onto an accelerator, the algorithm was implemented as a post-pass to compilation in the Trimaran toolset. The number of instructions needed to retarget each loop was recorded using OProfile [18] on an x86 system, which reads on-chip performance counters; the average translation penalty per loop is reported in Figure 8.

These penalties were measured while translating loops to target the specific LA proposed in Section 3. However, the results *are* representative of a broad class of accelerator architectures. ResMII, RecMII, and Priority calculation runtimes are a function of both the number of operations and data dependences in a loop, not the targeted architecture. The greedy CCA identification algorithm will select larger or smaller subgraphs based on the size of the targeted CCA, but the algorithm still selects each operation as a seed at most once and recursively grows that seed independent of the CCA architecture. If no CCA exists in the system, the overhead is simply eliminated. The register assignment process uses a one-to-one mapping from the baseline ISA to the accelerator registers; this is only architecture dependent to the extent that if there are too few registers, translation will abort. Scheduling is the only step where the overhead is highly architecture-dependent. Experiments show, however, that scheduling comprises less than 3% of the translation overhead. Even if the accelerator architecture made list scheduling 5 times longer (highly unlikely), this overhead would only constitute 12.5% of the total translation time. For these reasons, we believe the results presented here are applicable to a wide range of LA architectures.

There are a few important trends to take away from Figure 8. First, the average loop translation time varies widely from benchmark to benchmark. The primary reason for this is that the size of the loops also varies by a large factor, and larger loops usually require more work to modulo schedule. A secondary reason for the high variance is that the algorithm used in the priority calculation takes significantly more time if there are many recurrences in the loop. Applications that took the longest time to translate did not necessarily have the largest loops.

The most important take-away from Figure 8 is the distribution of time spent in various phases of the modulo scheduling algorithm. On average, it took approximately 99,716 instructions to map each loop onto the targeted LA. 69% of those instructions were devoted to calculating the priority used in scheduling, and 20% of the instructions were spent mapping subgraphs onto the CCA. The vast majority of translation time was spent performing these two tasks, which motivates us to perform these steps statically if possible. The remaining components of the algorithm only constitute 10,908 instructions on average, implying they can be done dynamically without causing significant performance degradation.

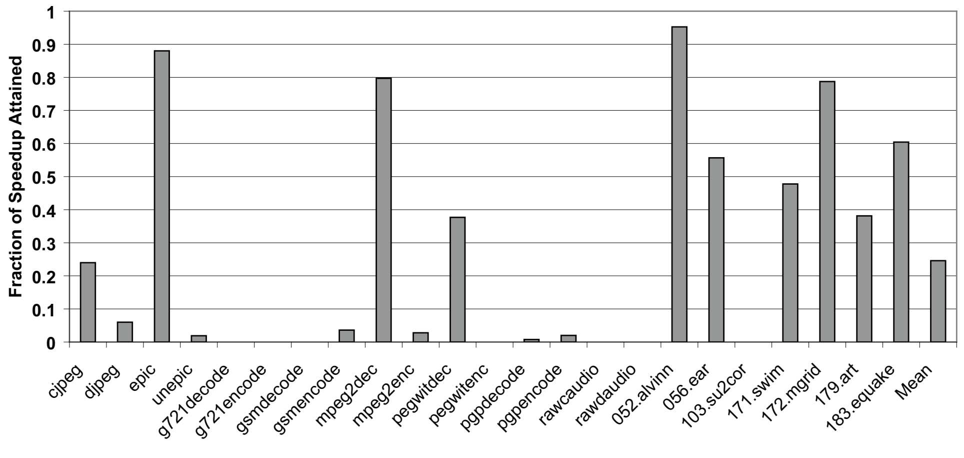


Figure 7. Percentage of speedup attained when using regular binaries compared to using binaries compiled using static loop transformation techniques (e.g., aggressive function inlining) that are too complex to perform dynamically.

Static ResMII and RecMII Calculation: Together, ResMII and RecMII calculation comprised roughly 1,250 instructions of translation overhead per loop. If it were necessary to reduce this overhead, each of these values could be calculated statically and placed in a data section in the binary right before the loop. Then once translation began, the VM could recover these values with two loads the addresses immediately preceding the top of the loop, forgoing the value calculation, and maintaining binary compatibility. The downside of statically determining the ResMII is that it is highly architecture dependent; an incorrect value would either produce a poor schedule (if ResMII was unnecessarily high), or cause scheduling to take much longer (if ResMII was too low) because of repeated attempts to schedule at impossibly low IIs. Static RecMII calculation makes more sense, because recurrence lengths generally do not change much as architectures evolve (e.g., an add in a recurrence path will typically takes 1 cycle no matter how the accelerator architecture changes). However, resource latencies *do* occasionally change, and the overhead needed to compute RecMII dynamically is quite low, thus we advocate performing both ResMII and RecMII calculations dynamically to maintain architectural independence.

Static Scheduling and Register Assignment: Scheduling and register assignment allotted for 9,650 instructions per loop of translation overhead. Performing either of these steps statically would strongly tie the application to one specific architecture, which is the antithesis of the desired outcome. Statically scheduling ties the binary to a specific quantity and latency for each type of execution resource (e.g., the architecture must have 3 2-cycle multipliers). Statically assigning registers ties the binary to a specific number of registers in the LA, as well as a specific configuration of the interconnect between FUs and register files. Statically performing either scheduling or register assignment reduces binary flexibility more than any of the other modulo scheduling steps, and so we propose performing them dynamically as well.

Static CCA Identification: CCA identification accounted for a significant fraction (20%) of the 100,000 instruction translation overhead. One potential way to statically encode this decision is *procedural abstraction*, proposed in previous work [5] and shown in Figure 9(b). Figure 9(a) shows the assembly instructions for the loop in Figure 5. Recall that in that example ops 5-6-8 were collapsed into a single CCA instruction. Statically a compiler can identify this subgraph and insert a branch-and-link instruction to a new function containing those ops.

```
   Loop:                Loop:                Data:
 1   Add             1   Add                   0
 2   Ld              2   Ld                    0
 3   Shl             3   Shl                   3
 4   Mpy             4   Mpy                   1
 5   And            16   Brl CCA               4
 6   Sub             7   Or                    2
 7   Or              9   Shr                   5
 8   Xor            10   Add                   6
 9   Shr            11   Add                   . . .
10   Add            12   Str                 Loop:
11   Add            13   Add              1   Add
12   Str            14   Cmp              2   Ld
13   Add            15   Br Loop          3   Shl
14   Cmp                 . . .            4   Mpy
15   Br Loop           CCA:              16   Brl CCA
     . . .           5   And              7   Or
                     6   Sub              9   Shr
                     8   Xor             10   Add
                                              . . .
    (a)                  (b)                  (c)
```

Figure 9. (a) Pseudo-assembly code for the loop in Figure 5. (b) How to statically encode CCA identification. (c) How to statically encode priority calculation

Then, the dynamic translator can recognize these simple function calls and attempt to map the instructions onto whatever CCAs are available in the LA. If a statically identified subgraph cannot be executed as a single unit on available CCAs, the ops can still be executed independently on the remaining execution resources. This property means static CCA identification does not tie the binary to one particular CCA (or even any CCA at all). Thus, performing CCA identification statically provides a significant reduction in mapping overhead without any compatibility impact.

Static Priority Calculation: Priority calculation is the longest step (by a significant margin) in modulo scheduling for the LA. Statically encoding the scheduling priority of loop operations in a binary compatible manner can be accomplished by placing a single number for each operation in a data section before the loop itself (shown in Figure 9(c)). With a single pass over the loop, the VM can determine how many ops

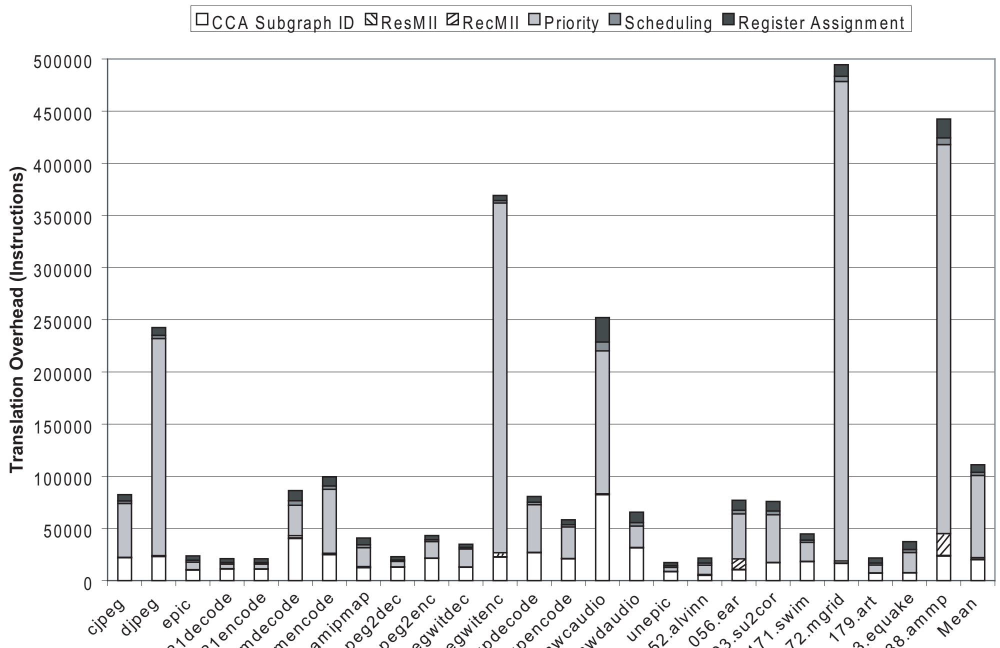


Figure 8. The measured translation penalty per loop.

are in the loop and determine the address of each op's priority by subtracting this number. For example, if a loop has 8 instructions, then an operation's priority value is located at $PC - 8 * instruction\_size$. In Figure 9, ops 1 and 2 have priority 0, because they are part of a memory stream. Ops 4 and 7 are on one of the critical recurrences, so they have priorities 1 and 2, respectively. The scheduler then uses these priorities to dynamically schedule the ops in a statically determined order.

Statically encoding the scheduling priority has the fortunate characteristic that it focuses on scheduling the most critical recurrences in the loop first, and recurrences are largely architecture independent.[3] Statically encoding priority in the binary enables a high quality schedule, while at the same time reducing the average loop translation time from 100,000 down to 31,000 instructions.

One potential, non-static solution to reducing the priority overhead is to use a simpler priority function. A promising candidate is the height-based priority function proposed in [24]. The simpler priority function was previously found to be effective in [24] because of the more exhaustive backtracking scheduler used in that algorithm, as opposed to the simpler list scheduling algorithm used here. However, using the height-based priority function in conjunction with the single-pass list scheduling often yielded sub-optimal schedules. Evaluation of this degradation is presented in Section 4.3.

A second potential non-static solution for reducing priority overhead is the combination of dynamically computing height-based priority combined with the complex scheduling algorithm from [24]. Priority computation would be simplified, but the scheduling is made more complicated to (hopefully) compensate for reduced priority quality. We chose not to investigate this combination in our experiments because previous work [7] already demonstrated that the modulo scheduling algorithm used here produces equivalent (or higher) quality schedules in less time than the scheduling algorithm used in [24]. There may be other scheduling algorithms that can rapidly produce high quality results using a simple priority function, but we leave their development to future work.

[3]It should be noted again that the criticality of recurrences are only architecture independent if execution latencies of the FUs remain consistent across the architectures (e.g., a multiplier is 3 cycles across different architectures).

4.3 Static/Dynamic Tradeoff Evaluation

Figure 10 shows the speedup for several different architectures over a single-issue processor modeled after the ARM 11 described in Section 3. These speedups are for the entire application, not just the loop bodies, and include communication overhead for transferring data to and from the processor over a system bus. The code cache used to store LA control provided enough space to store the previous 16 translated loops using an LRU eviction policy. Using the target architecture proposed in Section 3, this works out to approximately 48 KB of dedicated storage, which is small compared with typical code cache sizes [8]. Code cache hit rates for each application varied slightly, but all were very close to 100%. Communication overhead between the general purpose processor and the LA was assumed to be a fixed 10 cycles (same as the L2 cache access time), although this latency is largely irrelevant given the streaming nature of the target applications.

The left-most bar for each application shows the speedup from using the LA assuming no translation penalty. This is equivalent to the speedup of a statically compiled binary. The next bar, labeled "Fully Dynamic", shows the speedup when assuming a realistic translation cache and the penalties measured from performing the entire modulo scheduling algorithm dynamically. The "Fully Dynamic Height Priority" bar is also fully dynamic, but instead uses the simpler height-based priority function. The "Static CCA/Priority" bar represents the speedups when CCA mapping and more-complicated priority calculation are performed offline and encoded in the binary. The "2-Issue" bar shows the speedup of a dual-issue CPU modeled after the ARM Cortex A8 described in Section 3, and the "4-Issue" represents a hypothetical quad-issue Cortex A8 with larger L2 cache. Note that the ARM 11 w/ loop accelerator would consume approximately 8.25 mm^2 of die area, compared to 10.2 mm^2 for the 2-issue CPU and 14.0 mm^2 for the 4-issue CPU.

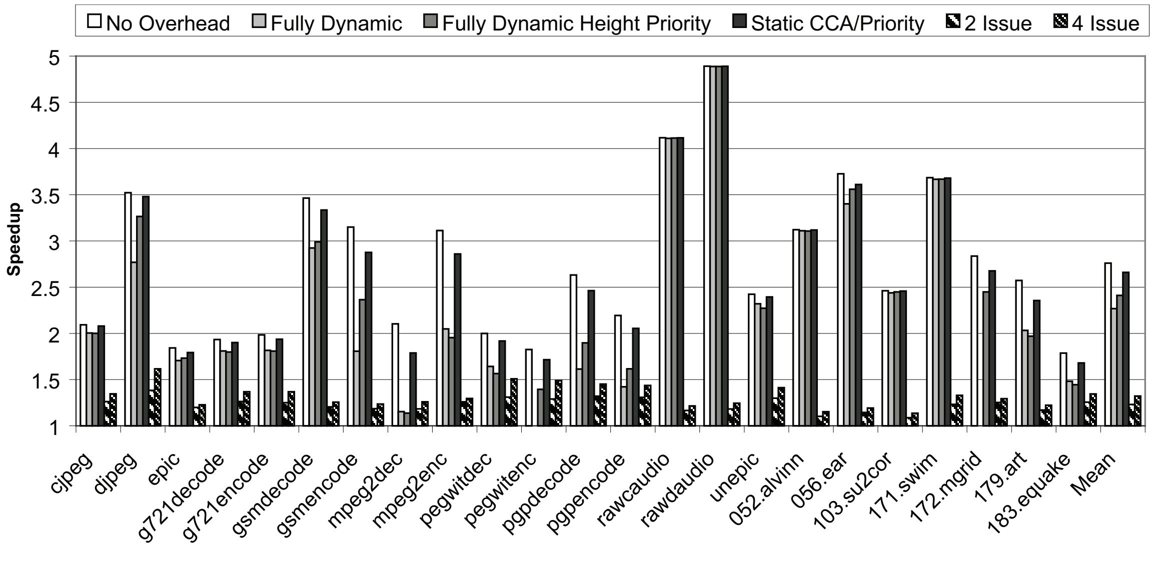


Figure 10. Static/dynamic and algorithm tradeoffs for key mapping stages.

Several interesting patterns emerge from Figure 10. First, for many benchmarks, such as rawcaudio, the translation overhead of performing modulo scheduling entirely dynamically has a negligible impact on the LA's speedup (comparing the first two bars). In the case of rawcaudio, there is only one critical loop in the application and so the translation cost is easily amortized. Other applications showed little performance degradation because their most critical loops were quite small, making the translation costs negligible. The translation overhead for many other loops *was* quite significant, however. Mpeg2dec notably went from a speedup of 2.1 down to 1.15, and pegwitenc and 172.mgrid lost all performance benefits from the LA. On average, factoring the translation costs brought the speedup from 2.76 down to 2.27.

The "Fully Dynamic" and "Fully Dynamic Height Priority" bars for each application show the tradeoff involved in using the more-complex priority function in comparison with the simpler height-based priority. The less sophisticated height-based priority function sometimes generates schedules with higher IIs (and thus, worse performance), but the translation times are significantly faster. On average, the benefits of faster translation time outweighed the benefits of better schedules, providing a speedup of 2.41 compared with 2.27.

The "Static CCA/Priority" bar in Figure 10 shows that by moving the particularly difficult portions of mapping loops offline, the speedups can approach that of natively compiled code. On average, performing CCA mapping and priority calculation offline reduced translation penalties to the point where the average speedup was 2.66 as compared with 2.76 for natively compiled code. This hybrid static/dynamic mapping strategy provides a significant 25% and 39% more speedup up over fully dynamic solutions utilizing height-based and recurrence-based priority functions, respectively.

The "2 Issue" and "4 Issue" bars in Figure 10 show that the loop accelerator is a much more effective use of die area than more general purpose processor enhancements, such as increasing issue width or cache size.

5 Related Work

As mentioned in previous sections, accelerators are a popular method to increase the performance and efficiency of microprocessor designs. Several people have proposed accelerators specifically targeting loop nests, because the regular control structure in loops provides significant efficiency gains over processors designed for general purpose control structures. The Reconfigurable Streaming Vector Processor (RSVP) [3] is a vector-based accelerator designed for loops in multimedia applications running in an embedded environment. The architecture is similar to what we have proposed; however, RSVP uses SIMD execution units, and a single SRAM to buffer memory accesses. Mathew et al. propose another LA architecture in [20], which is very similar to the architecture proposed here. The main difference is the memory buffering structure and type of execution resources provided. This paper extends these two previous works by providing a quantitative analysis of accelerator resource needs using loops from a diverse application set. Other work, such as [21, 25], proposed adding hardware to a standard pipeline for efficiently supporting the control structures of loops. The control in our proposed accelerator design is very similar to [21], but this work extends prior customizations by additionally customizing execution and memory resources. The LA architecture presented in this paper was primarily developed to provide a realistic target for evaluating dynamic mapping algorithms.

Statically generating efficient code for loops is also an area of much related work. Software pipelining has proven to be an excellent way to improve the resource utilization of loop execution. Lam [16] showed that developing an optimal software pipelining is an NP-complete problem, and so many heuristics have been developed to produce high-quality schedules in a reasonable amount of time [7, 17, 19, 24, 25, 26]. The most pertinent related work is the Swing Modulo Scheduling algorithm [19], the basis of our analysis in Section 4.2. Later work [7] demonstrated that this algorithm produces high quality schedules in much shorter runtimes than other modulo scheduling algorithms, making it a good starting point for dynamically retargeting loops. While the work in this paper did not exploit this fact, modulo scheduling algorithms have been extended to support loops with complex control flow, such as side exits [17], and entire loop nests [26], not just innermost loops. One major contribution of this paper is the evaluation of modulo scheduling in the context of dynamically targeting an LA. The relative runtime of each modulo scheduling stage was measured, and we explore the tradeoffs associated with statically encoding the results of each stage in an application binary.

Abstracting the underlying hardware structure to enable hardware innovation without affecting binary compatibility also has much related work. Perhaps the best known example of this is the Transmeta Code Morphing Software [8], which dynamically converts x86 applications into VLIW programs. DynamoRIO [2], Daisy [9, 10], and DIF [22] are all examples that dynamically translate applications to target entirely different microarchitectures. Several proposals exist to only translate select portions of an application to target accelerators. For example, [5, 14] all explored the benefits of dynamically binding applications to acyclic accelerators. Other work [6] looked at dynamically binding for SIMD accelerators. This paper is the first proposal for dynamically binding to cyclic accelerators.

6 Concluding Remarks

Adding customized hardware to a processor is an effective way to improve the performance and efficiency of the system. However, significant hardware and software non-recurring engineering costs prevent customized hardware from being adopted in many situations. This paper addresses those costs in the context of cyclic computation. Cyclic computation accelerators are a compelling design point, because they encompass a larger fraction of many applications' execution time than acyclic accelerators, even though cyclic accelerators are not as broadly applicable as acyclic ones.

This paper presented the design of a generalized loop accelerator. Design space exploration was used to ensure that the accelerator is applicable to a wide range of media and floating point applications. This generalized design provides a good architecture for executing common modulo schedulable loops, thus eliminating the engineering costs associated with designing loop-specific accelerators from scratch. The proposed design provides 83% of the speedup attainable through a hypothetical accelerator with infinite resources, and consumes only 3.8mm^2 of die area in a 90nm process.

Software costs were addressed by virtualizing the accelerator interface. Modulo schedulable loops are statically transformed in the binary and expressed in the baseline instruction set. At runtime, a dynamic translator attempts to map the loops onto any available accelerators using modulo scheduling. This work found dynamically modulo scheduling loops has a significant performance overhead and proposed statically encoding scheduling priority and CCA mapping to be an effective technique for minimizing the overhead. Overall, the loop accelerator and dynamic compilation system provided a mean speedup of 2.66 over a single-issue processor, and the resulting binary remains flexible enough to be used by systems with different (or even no) accelerators.

7. Acknowledgments

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. We owe thanks to Krisztian Flautner, Todd Austin, Trevor Mudge, and Serap Savari who helped shape this work. This research was supported by ARM Limited, the National Science Foundation grants CNS-0615261 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] S. Aditya and M. Schlansker. ShiftQ: A buffered interconnect for custom loop accelerators. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 158–167, Nov. 2001.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[3] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 141–150, 2003.

[4] N. Clark et al. OptimoDE: Programmable accelerator engines through retargetable customization, Aug. 2004. In *Proc. of Hot Chips 16*.

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

[6] N. Clark et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 216–227, 2007.

[7] J. Codina, J. Llosa, and A. Gonzalez. A comparative study of modulo scheduling techniques. In *Proc. of the 2002 International Conference on Supercomputing*, pages 97–106, June 2002.

[8] J. Dehnert et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[9] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–38, June 1997.

[10] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, June 2001.

[11] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, Nov. 2005.

[12] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for cell broadband engine system software. *IEEE Computer*, 40(6):37–47, 2007.

[13] A. Hormati et al. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 341–353, Mar. 2007.

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

[15] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 277–288, 2006.

[16] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–327, 1988.

[17] T. Lattner. An Implementation of Swing Modulo Scheduling with Extensions for Superblocks. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, June 2005.

[18] J. Levon. *OProfile - a System Profiler for Linux*, 2004. http://oprofile.sourceforge.net/doc/index.html, Retrieved June 6, 2007.

[19] J. Llosa et al. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. of the 5th International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[20] B. Mathew and A. Davis. A loop accelerator for low power embedded VLIW processors. In *Proc. of the 2004 International Conference on on Hardware/Software Co-design and System Synthesis*, pages 6–11, 2004.

[21] M. Merten and W.-M. Hwu. Modulo schedule buffers. In *Proc. of the 34th Annual International Symposium on Microarchitecture*, pages 138 – 149, 2001.

[22] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[23] U. Nawathe et al. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara2), Feb. 2007. In *Proc. of ISSCC*.

[24] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[25] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation for modulo scheduled loops. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, Nov. 1992.

[26] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Transactions on Architecture and Code Optimization*, 4(1):7, 2007.

[27] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[28] M. Sivaraman and S. Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 35–42, 2002.

[29] J. E. Smith and R. Nair. *Virtual Machines*. Morgan Kaufmann Publishers, 2005.

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