Bridging the Computation Gap Between Programmable Processors and Hardwired Accelerators

Kevin Fan[*] Manjunath Kudlur[†] Ganesh Dasika Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{fank, kvman, gdasika, mahlke}@umich.edu

Abstract

New media and signal processing applications demand ever higher performance while operating within the tight power constraints of mobile devices. A range of hardware implementations is available to deliver computation with varying degrees of area and power efficiency, from general-purpose processors to application-specific integrated circuits (ASICs). The tradeoff of moving towards more efficient customized solutions such as ASICs is the lack of flexibility in terms of hardware reusability and programmability. In this paper, we propose a customized semi-programmable loop accelerator architecture that exploits the efficiency gains available through high levels of customization, while maintaining sufficient flexibility to execute multiple similar loops. A customized instance of the loop accelerator architecture is generated for a particular loop and then the data and control paths are proactively generalized in an efficient manner to increase flexibility. A compiler mapping phase is then able to map other loops onto the same hardware. The efficiency of the programmable accelerator is compared with non-programmable accelerators and with the OpenRISC 1200 general purpose processor. The programmable accelerator is able to achieve up to 34x better power efficiency and 30x better area efficiency than a simple general purpose processor, while trading off as little as 2x power and area efficiency to the non-programmable accelerator.

1. Introduction

In the coming years, the deployment of untethered computers will increase rapidly. The prime example of such a device is currently the cell phone, but in the near future we expect to see the emergence of new classes of such devices. These devices will improve on the mobile phone by moving advanced functionality, such as always-on Internet access, human-centric interfaces with voice recognition, and high-definition video coding, into the device. These new devices will require higher throughput and more energy-efficient computer systems to meet application performance requirements, while still operating for long periods of time on a battery. For example, the projected data rates for 4G wireless data communication are expected to increase 50 times over current 3G standards [27].

These performance and energy demands are in direct conflict with an increasingly important characteristic, post-programmability. High performance and low power can often be achieved using hardwired solutions, e.g., application-specific integrated circuits or ASICs. Modern embedded systems generally employ ASICs for the most compute-intensive tasks. However, a programmable solution offers several key advantages. First, software implementations allow the application to evolve in a natural way after the chip has been manufactured due to changes in the specification, bug fixes, or the addition of new features. Second, multi-mode operation is enabled by running multiple different applications or variants of applications on the same hardware. Third, time-to-market of new devices is lower because the hardware can be re-used. And finally, chip volumes are higher as the same chip can support multiple products in the same family.

The tradeoffs between performance, power, and programmability are at the heart of the hardware implementation choice that designers are forced to make. ASICs provide the highest performance and lowest energy solutions for specific problems. However, they offer little in the areas of programmability and hardware re-use due to the hardwired nature of the design. At the other end of the spectrum are processors and DSPs. Processors offer full programmability and thus the ability to execute a wide range of applications. But, processors offer poor energy efficiency and often cannot meet application performance requirements. ASICs typically offer 100-1000x more energy-efficiency for specific applications than processors. Middle-ground solutions offer the promise of high efficiency together with full programmability. However, they often fall short of these goals. For instance, FPGAs achieve extremely high performance for bit-level parallel computation. But, the overhead of gate-level reconfigurability often causes them to fall short in applications that have limited parallelism or rely on more expensive computations, such as multiplies. Application specific instruction-set processors (ASIPs) also compromise between the flexibility of a processor and the efficiency of an ASIC by introducing customizations for specific applications. However, they generally retain a high level of programmability and do not approach the efficiency of ASICs.

A key question that this paper investigates is: *How much*

[*]Currently with Parakinetics, Inc.
[†]Currently with NVIDIA Corporation.

programmability is really required in a design? Programmability is generally thought of as a binary issue - either a design is programmable or not. Programmable designs support a wide range of applications while hardwired designs support a single algorithm implementation. An important insight is that semi-programmable solutions may be enough for many embedded designs. For example, video coding standards are typically developed years ahead of time by industrial consortiums [14]. These standards go through many rounds of development and adjustment, but the core algorithm kernels often evolve at a relatively slow rate. At the same time, domain-specific hardware is often essential to achieve the necessary performance and energy efficiency. And, this customized hardware is neither appropriate nor efficient for applications outside the domain. Therefore, providing universal programmability may have little practical value.

Our approach is to push programmability into a highly customized hardware substrate to retain the high performance and energy efficiency of an ASIC, while offering a limited degree of post-programmability. The starting point is a stylized loop accelerator (LA) that is customized for a single application loop nest [23, 9]. The LA is a direct hardware realization of a modulo scheduled loop [22]. Each LA has a specialized datapath, including function units, register files, and interconnect, and a simple controller driven by the initiation interval of the schedule. We generalize the structure of the base LA template to create a semi-programmable solution, termed a programmable LA or PLA. However, the PLA datapath is still highly specialized with point-to-point interconnect, fixed-capability function units, and limited storage to retain its inherent efficiency characteristics. Such a platform cannot execute an arbitrary loop. Rather, the programmability objective is to map loops with similar computation structure onto a common hardware platform, such as two loops from the same application domain or a single loop that has undergone small to modest changes in composition.

This paper offers the following contributions:

- An analysis of the evolution of several media applications to understand the programmability needs of customized hardware.
- A parameterized template for a PLA is developed. The template offers high degrees of customization to the target loop, while providing programmability for a range of loops with similar computation structure.
- The performance, power efficiency, and programmability of the resultant PLAs are evaluated and compared to single-function LAs and the OR-1200 embedded processor for a range of compute-intensive kernels.

2. Motivation

2.1. Architecture Style vs. Efficiency

A wide range of architectures have been designed before to address the problem of providing high performance computation efficiently. These solutions maintain or sacrifice programmability to various degrees depending on the domain they target. This section describes some of these solutions and

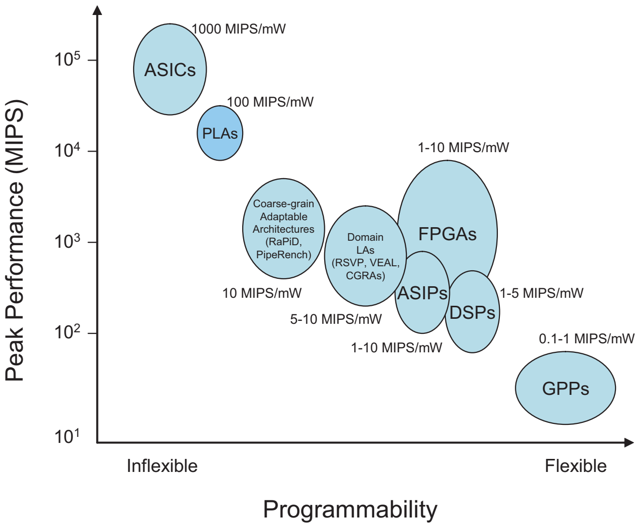


Figure 1: Comparison of peak performance, power efficiency, and programmability of different architecture design styles.

motivates the need for our semi-programmable accelerators. Figure 1 shows the peak performance achievable by different architecture styles and their programmability and power efficiency. The x-axis in Figure 1 indicates programmability of different solutions. General purpose processors (GPPs) which fall on the lower right corner of the figure are highly programmable solutions, but are limited in terms of the peak performance they can achieve. Also, structures like instruction decoders and caches that are needed to support programmability consume energy, resulting in very low computation efficiency of about 1 MIPS/mW for Pentium M. On the other extreme of the spectrum are ASICs. ASICs are custom designed for a particular problem, without extraneous hardware structures. Thus, ASICs have high computational density with hardwired control, resulting in high computation efficiency up to 1000 to 10000 times more than GPPs. The space between these two extremes is populated by different solutions that have varying degrees of programmability.

Digital signal processors [25, 26, 18] increase computation efficiency by providing specialized features that optimize execution of signal processing algorithms. These features include special arithmetic operations like multiply-accumulate and bit manipulation operations, hardware modulo addressing, and memory architecture optimized for streaming data. A wide range of DSP algorithms can be executed efficiently on these processors efficiently. DSPs typically offer an order of magnitude increase in power efficiency.

Application specific instruction-set processors (ASIPs) are processors with custom extensions for a particular application. They can be quite efficient when running the applications for which they are designed, and they are also capable of running any other application, though with reduced efficiency. Examples include Tensilica [24], ARC [1], and Custom-Fit Processors [11]. Transport triggered architectures (TTAs) [5] define another template for ASIPs. Their basic design resembles that of a VLIW processor. However, TTAs expose more of the microarchitecture, namely, the internal transport buses, to the compiler. The efficiency of TTAs is increased by moving values directly between function units, thus reducing register file accesses. The flexibility of TTAs is limited because of the complex scheduling decisions needed in the compiler. Computationally intensive code with well defined memory access patterns map well to TTAs. MOVE32INT [6] and MAXQ [16] are examples of instantiations of TTA.

Domain loop accelerators are designed to execute computation intensive loops present in media and signal processing domains. Their design is close to a VLIW processor, but with a much higher number of function units, and thus higher peak performance. Very long instruction words in a control memory direct all function units every cycle. However, domain LAs have less flexibility than GPPs because only highly computationally intensive loops map well to them. Some examples of architectures in this design space are VEAL [4], RSVP [3], CGRAs [17, 21], and [15].

FPGAs have fine grain logic blocks that can be reconfigured to perform various bit level logic and arithmetic functions. The fine grain reconfigurability allows FPGAs to be very flexible. Bit parallel computations in domains like encryption can be performed very efficiently. However, complex integer and floating point operations do not map well on to FPGAs. Thus, for some domains, FPGAs are very flexible and highly efficient.

Coarse-grain adaptable architectures have coarser grain building blocks compared to FPGAs, but still maintain bit-level reconfigurability. The coarser reconfiguration granularity improves the computation efficiency of these solutions. However, non-standard tools are needed to map computations onto them and their success have been limited to the multimedia domain. PipeRench [12], RaPiD [8] are some examples of coarse-grain adaptable architectures.

The programmable solutions shown in Figure 1 are all "universally" programmable, allowing any loop to be mapped on to them, although at varying degrees of efficiency. There is a wide gap between the efficiency that can be achieved by ASICs and the efficiency that can be achieved by these programmable solutions. Section 2.2 shows that there are instances where there is a narrow requirement of flexibility. Using any of these above solutions is a overkill for these instances as these solutions sacrifice too much efficiency for the needed flexibility. We position our PLAs in the design space where non-trivial amount of programmability as well as the the high efficiency of ASICs are required.

2.2. Programmability Case Study

As applications evolve over time, code changes are inevitable. Whether due to changing requirements, evolving standards, bug fixes, or new features, software is constantly in flux. With hardwired solutions, every time the code in an accelerated loop changes, new hardware must be synthesized even if the changes are small and the dataflow between operations within the loop is substantially similar. By adding some programmability, the hardware can be made robust in the face of such changes. By looking at some loops from real applications, we can get a feel for what kinds of changes typically occur.

Figure 2 shows a loop from the `faad2` application, which is a commonly used free audio decoder for the Advanced Audio Coding (AAC) standard. The figure shows that between revisions 1.39 and 1.40 of the software, the loop has been modified with the addition of an if-clause, while the rest of the loop remains the same. This represents the addition of a new feature that requires certain new code in the loop to be guarded under a flag. To implement the body of the if-clause, the hardware must have function units capable of performing load, multiply, and store. As these operations are already present elsewhere in the loop, the new code should ideally be executable on the same hardware, although the level of performance may be lower because the same hardware resources are being used to execute more operations. The additional control flow should not present a problem because the loop can be if-converted, and a compare operation is not required inside the loop because the if-condition is live-in.

Version 1.39

```
for(k=0; k<N4; k++) {
  ...
  real = Z1[k][0];
  img  = Z1[k][1];

  Z1[k][0] = real * sincos[k][0]
           - img*sincos[k][1];

  Z1[k][0] = Z1[k][0] << 1;
}
```

Version 1.40

```
for(k=0; k<N4; k++) {
  ...
  real = Z1[k][0];
  img  = Z1[k][1];

  Z1[k][0] = real * sincos[k][0]
           - img*sincos[k][1];

  Z1[k][0] = Z1[k][0] << 1;
  if(b_scale) {
    Z1[k][0] = Z1[k][0] * scale;
  }
}
```

Figure 2: Feature addition to `mdct.c` in `faad2`.

Version 1.33

```
for(k=0; k<N4; k++) {
  ...
  uint16_t n = k << 1;
  ComplexMult(...);

  X_out[          n] =  RE(x);
  X_out[N2 - 1 - n] = -IM(x);
  X_out[N2 +      n] =  IM(x);
  X_out[N  - 1 - n] = -RE(x);
}
```

Version 1.34

```
for(k=0; k<N4; k++) {
  ...
  uint16_t n = k << 1;
  ComplexMult(...);

  X_out[          n] = -RE(x);
  X_out[N2 - 1 - n] =  IM(x);
  X_out[N2 +      n] = -IM(x);
  X_out[N  - 1 - n] =  RE(x);
}
```

Figure 3: Bug-fix to `mdct.c` in `faad2`.

Figure 3 shows another loop from the same application. In this case, the code changes from version 1.33 to 1.34 consist of sign changes on the right hand side of some assignment statements, as might occur in a bug fix. These sign changes correspond to dataflow changes in the loop, as some values now must go through a subtractor, while other values should no longer go through a subtractor. Alternatively, the dataflow changes could occur post-negation, with the same values being stored to different addresses. In either case, the number of operations does not change, but the communication between operations changes, and the hardware should be flexible enough to accommodate this.

It can be seen that loops in real applications undergo minor changes over time. Typically, the bulk of the computation in the loop remains the same, but small changes need to be made to fix bugs or implement new features. Since the changes do not alter the loops significantly, it is possible to design custom hardware to accelerate the original loop, supporting just enough programmability to continue to accelerate the loop efficiently as the source code evolves.

3. Single-function Accelerator

A single-function LA is used as a baseline in this paper. This accelerator is designed to execute a specific loop at a given performance level, and is not programmable. Then, starting from

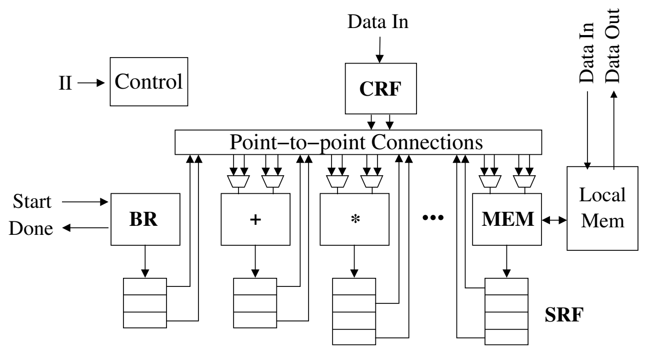


Figure 4: Template for single-function loop accelerator.

the single-function baseline, the datapath is generalized to create a more programmable design. The goal is to remove or relax the features of the architecture that are most limiting in terms of programmability, while retaining the efficiency available through customization. In this section, the architecture for the single-function accelerator is described. The datapath generalizations are described in the next section.

3.1. Accelerator Design

Figure 4 shows the hardware schema for the single-function LA [23, 9]. The LA is designed to efficiently implement a modulo schedule in hardware. Modulo scheduling is a method of overlapping iterations of a loop to achieve high throughput [22]. The performance of the schedule is determined by the *initiation interval* (II), or the number of cycles between successive iterations of the loop; thus, a lower II corresponds to higher throughput. The modulo schedule contains a *kernel* that repeats every II cycles and may include operations from multiple loop iterations.

The LA is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU performs a specific set of functions that is tailored for the particular loop. Each FU writes to a dedicated shift register file (SRF); in each cycle, the contents of the registers shift downwards to the next register. Wires from the registers back to the FU inputs allow data transfer from producers to consumers. Multiple registers may be connected to each FU input; a multiplexer (MUX) is used to select the appropriate one. Since the operations executing in a modulo scheduled loop are periodic, the selector for this MUX is essentially a modulo counter. In addition, a central register file (CRF) holds static live-in register values that cannot be stored in the SRFs.

The schema described is a template that is customized for the particular loop being accelerated. The number, types, and widths of the FUs, the widths and depths of the SRFs, and the connections from the SRFs to the FUs are all determined from the loop. During synthesis, the loop is first modulo scheduled to meet a given performance requirement, and then the details of the LA datapath are determined from the communication patterns in the scheduled loop.

The control path for the single-function LA consists of a finite state machine with II states corresponding to each of time slots in the kernel of the modulo schedule. In each state, control signals direct the execution of FUs (for FUs capable of multiple operations) and control the MUXes at the FU inputs.

Figure 5(a) shows a portion of the loop from the FIR filter application. Assuming the given II is 2, the LA will have two adders, one memory unit, and one multiplier. When the operations in the loop are scheduled as shown in Figure 5(b), the resulting single-function LA hardware will be as shown in Figure 5(c). The connectivity within the LA is limited because only those connections required to support this schedule are created. For example, data for the multiplier can only come from the memory unit.

Now, assume that a second loop (Figure 5(d)) is to be mapped to the same LA. This second loop is somewhat similar to the first, in that it also contains adds, loads, and multiplies. However, the functionality is different, and the communication patterns between operations are different as well. The next subsection discusses the LA features that make it difficult to map the second loop onto the LA.

3.2. Limitations to Programmability

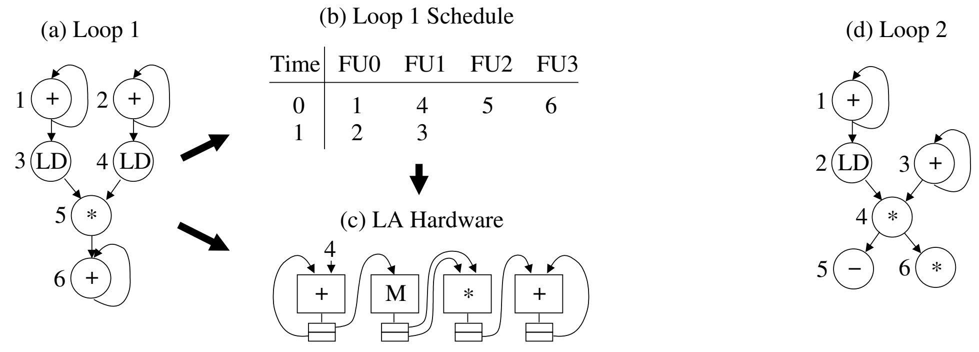
Since an LA is designed for a specific loop, its datapath is customized for the specific computation and communication needs of the scheduled operations in that loop. In fact, this is how the LA gets its efficiency wins: point-to-point connections, limited storage, and customized functionality. However, the same datapath features that lead to an efficient LA also restrict its flexibility:

Point-to-point connectivity. A major area where the LA achieves efficiency wins is the point-to-point connectivity scheme. Only those connections that are needed to sustain the producer-consumer communications in the modulo schedule exist in the single-function LA. This allows the number of execution units to be scaled up, but not all FUs are able to communicate directly with other FUs, making it difficult to map new applications onto the hardware. Furthermore, the connections are port-specific, so even if two FUs are able to communicate with each other, it may not be possible to route a value from the correct producer port to the correct consumer port. In Figure 5, Loop 2 contains a dataflow edge from an ADD (operation 3) to a MUL (operation 4), which did not exist in Loop 1, thus there is no corresponding connection in the LA datapath.

Shift register files. The nature of the SRFs limits the flexibility of the hardware. Because they have a fixed number of entries, any value produced by the corresponding FU must be consumed within a certain number of cycles, or it will "fall off" the end of the SRF. In addition, the read and write ports of the SRFs are not addressable. Instead, specific SRF entries are connected to consuming FUs, so the values can only be read at certain times. In the example of Figure 5, the multiplier has two registers at its output because its result is read two cycles after it is computed. Only the second register can be read, because Loop 1 requires just this. Thus, Loop 2 cannot be scheduled such that the result of a multiply is used in the next cycle.

Functionality. The opcode repertoire of each FU is customized for a given loop. If, for example, an LA is built for a loop that does not contain any shift operations, no FUs will be capable of performing shifts. If a new loop contains a shift operation, it will not be possible to map the new loop onto the LA. In the example of Figure 5, the hardware contains no subtractors because Loop 1 did not contain any subtract operations; thus, Loop 2 cannot be mapped onto the LA.

To understand these limitations more generally, Figure 6



Time	FU0	FU1	FU2	FU3
0	1	4	5	6
1	2	3		

Figure 5: LA scheduling and synthesis example.

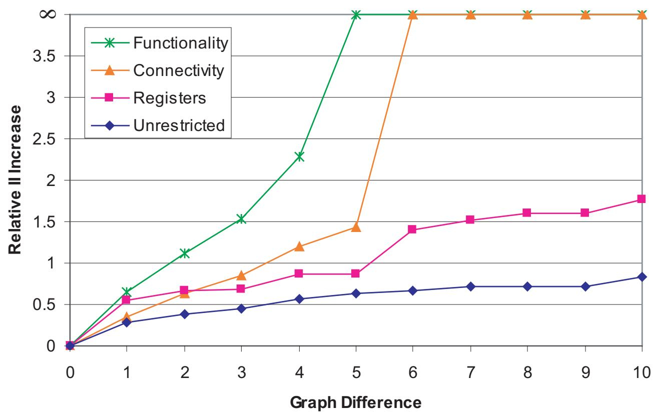


Figure 6: Average performance degradation resulting from LA datapath restrictions.

shows the effects of these datapath restrictions on the programmability of LAs created for several compute-intensive loops.[1] A number of synthetic loops are generated by randomly applying transformations to each initial loop. As more transformations are applied, the loop becomes progressively more different from the original loop, and it becomes harder to map the loop onto hardware designed for the original loop. The number of transformations (and hence a measure of difference from the original loop) is shown along the x-axis. When a loop is too different from the original loop, scheduling will fail at a certain II; the II is then incremented and scheduling is attempted again. Often, scheduling succeeds with a larger II because of the greater number of scheduling slots available. The y-axis shows the amount of this performance degradation in terms of II increase.

The graph shows four curves. The lowest curve, labeled "unrestricted," is the baseline: a generalized loop accelerator without the datapath restrictions described previously. II increases are primarily due to resource limitations; the unrestricted datapath has a fixed number of general FUs, but may not sustain the original II as more operations are added to the loop via the transformations. The curve labeled "registers" represents the LA with limited-size SRFs, but no other restrictions. Similarly, "connectivity" is the LA with point-to-point, port-specific connections and no other restrictions, and "functionality" is the LA with customized FUs but no other restrictions. In the latter two cases, the high II increase of infinity indicates that scheduling failed for all of the test cases at all IIs, due to either unsupported functionality or connectivity.

It can be seen that although these datapath features allow the LA to be very efficient, they limit its programmability. Mapping a new loop onto an LA can easily result in performance degradation or failure when the new loop modestly differs from the original loop. In particular, Figure 6 shows that the largest degradations, or even failures, are caused by the limited functionality and connectivity of the datapath. The next subsection will discuss the changes made to the LA datapath to improve its programmability and better support the execution of other loops.

[1]See Section 5 for details on the experimental methodology.

4. Programmable Loop Accelerator

4.1. From Single-function LA to Programmable LA

To build a programmable loop accelerator (PLA), the datapath features of the single-function LA that are least flexible should be generalized in a power and area efficient manner. Table 1 summarizes the key datapath characteristics of the PLA. The rest of this section will describe these in more detail.

Generalized functionality. The LA is limited by the opcodes supported by the FUs. FUs can be generalized with low additional cost by adding functionality that is complementary to existing functionality. For example, any adder can be generalized to support both addition and subtraction with low additional cost. Other generalizations include broadening the opcode repertoire of logical, memory, comparison, and shift FUs to include all variants of those respective opcodes (e.g. all shift FUs are expanded such that they are capable of left and right arithmetic and logical shifts). The costs of FU generalization include increased hardware area and power consumption, as well as increased encoding requirements for the larger number of supported opcodes. For the example second loop of Figure 5, there is a subtract operation that is not supported by the single-function LA. By generalizing the adders to adder-subtractors, the functionality of the second loop will be supported.

Global connectivity. Two techniques are used to relax the constraints of point-to-point connectivity. First, all FUs are given the ability to perform a MOV; that is, copy one of its inputs to its output. This allows values to be transferred from a source FU to a destination FU via intermediate FUs. Second, a low-bandwidth bus is created that connects all FUs in the accel-

Restriction	LA Characteristic	Application Change to Support	PLA Approach
Functionality	Custom FU repertoire	New opcodes	Low-cost FU generalization
Connectivity	Point-to-point connections	New communication patterns	MOVs, Low-bandwidth bus
	Port-specific connections	Swap input operands	FU input MUXes
Storage	Limited size	Longer variable lifetimes	Rotating register files
	No addressability	New communication patterns	
Control	Staging predicates	Operations in different stages	Predicate bus
	Hardwired control	Any change	Control memory
	Hardwired literals	Different literals	Literal file

Table 1: Summary of PLA architecture changes to address programmability limitations of the single-function LA.

erator.[2] This allows a single value transfer from any FU to any other FU each cycle. The bus is scheduled by the compiler and thus is not arbitrated. Such a global bus can be viewed as a fall-back communication path, ensuring that communication from any FU to any other FU is possible. Thus, the programmability of a given accelerator (in terms of the number of different loops that can be mapped onto it) increases significantly; however, since the bus is low bandwidth, if a loop requires a large number of bus transfers, it will not be possible to achieve a schedule with low II (high performance).

The global bus incurs additional hardware cost as each register file contains a new read port which can place a value onto the bus, and each MUX contains a new input which allows the FU to read the value from the bus in addition to the existing point-to-point connections.

In Loop 2 of Figure 5, two of the communication paths are not supported by the single-function LA. Specifically, the edge from operation 3 to operation 4 cannot be mapped onto the LA because there is no wire from an adder to a multiplier, and the edge from operation 4 to operation 6 cannot be mapped because there is no wire from a multiplier to a multiplier. The $3 \rightarrow 4$ communication can be handled by inserting a MOV to pass the value from the adder through the memory unit to the multiplier. The $4 \rightarrow 6$ communication can be handled by passing the value on the global bus.

Figure 7 shows the results of the datapath generalization so far (registers have been omitted from the hardware diagrams for clarity). FUs have been generalized, MOVs are supported, and a global bus has been added. Loop 2 is now able to execute on the LA originally designed for Loop 1, using the II=2 schedule shown. The remainder of this section discusses additional datapath restrictions that are not shown in this example.

Port swapping. In the single-function LA, each input port of an FU has its own connections to specific register files. To schedule another operation onto that FU, both of the operation's source operands must be routable from where they are produced to the corresponding input ports of the FU. Scheduling can fail if either routing is not possible. If the operation is commutative, then swapping the sources of the operation may result in a successful schedule; however, to relax this constraint more generally, the actual physical connectivity within the datapath should be increased. One way of accomplishing this is to introduce an additional level of MUXing at the FU input ports such that the ports can swap values. However, modeling this two-level MUX is challenging for the compiler, as it must ensure during scheduling that invalid combinations do not occur. Thus, a more general strategy is to widen the input MUXes to allow each input port to read its operand from any connection originally made to either port.

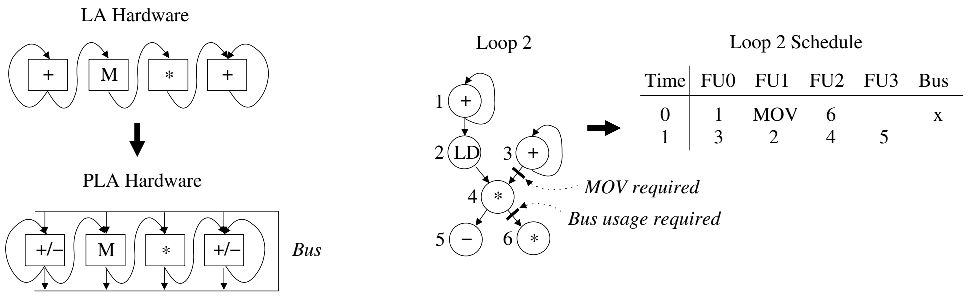
[2]This bus may be pipelined or organized in a hierarchical manner for larger accelerators.

Rotating register files. The flexibility-limiting aspects of the SRFs can be addressed by replacing them with rotating register files (RRFs) [7]. RRFs are similar to standard addressable register files, with the extension that the physical register address is a function of the input address and a base register, which is decremented once per iteration. RRFs are well suited for modulo scheduled loops because this renaming mechanism overcomes cross-iteration register overwrites. The replacement of SRFs by RRFs improves programmability, but it introduces some additional hardware, namely base registers, adders, and decoders for the read and write ports. In addition, the sizes of the RRFs are rounded up to the next power of two to facilitate efficient implementation of register rotation. However, the RRFs remain small (thus the width of base registers and adders is only a few bits per register file) and distributed.

An additional cost of replacing SRFs with RRFs is in the control path: each read and write port now requires an address, whereas the hardwired SRFs required no addressing at all.

Global staging predicate. The LA is a hardware implementation of a modulo scheduled loop; as such, operations in the loop kernel are scheduled in various stages, and must be controlled by guarding predicates as the software pipeline fills and drains. This guarding predicate is produced by the branch unit and consumed by all other FUs. In the single-function LA, specific connections are made between registers in the branch unit's output SRF and the other FUs. This effectively restricts the stage in which operations on a given FU may be scheduled. To generalize this aspect of the hardware, staging predicates are broadcast over a bus to all FUs, significantly increasing scheduling flexibility. The additional cost is low because each predicate is a single bit, and the number of predicates required is just the number of stages in the schedule.

Control memory. In the single-function LA, the datapath is directed by hardwired control signals generated by a finite state machine. To allow programmability, the datapath should instead be directed by signals from a control memory. The size of the control memory depends on the number of FUs, MUXes, and registers in the design as well as on the maximum allowed II. In addition, in the single-function LA, literal operands are hardwired. Clearly, this does not allow a loop with different literals to be mapped to the hardware. By placing literals into a central literal file, different literal values may be used for different loops.



Time	FU0	FU1	FU2	FU3	Bus
0	1	MOV	6		x
1	3	2	4	5	

Figure 7: PLA generalization and scheduling example.

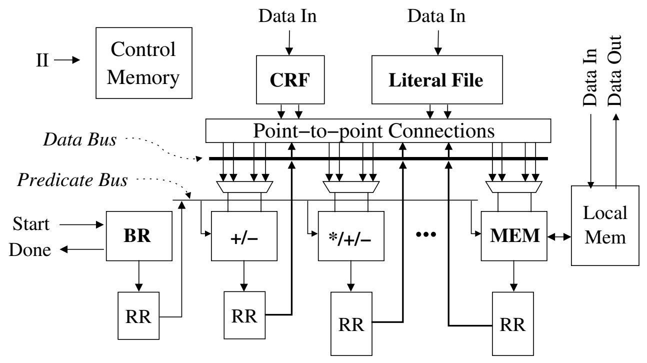


Figure 8: Template for programmable loop accelerator.

4.2. PLA Architecture Template

Figure 8 shows the template for the PLA, generalized from the datapath shown in Figure 4. The accelerator is designed for a specific loop at a specific throughput, but contains a more general datapath than the single-function LA to allow different loops to be mapped onto the hardware. FUs have been generalized to support more functionality; a low-bandwidth bus connects all FUs; the staging predicate is broadcast over a bus; SRFs are replaced with small, distributed RRFs; and the FU input MUXes are widened. The area and power overheads of these changes will be discussed in Section 5.

The loop accelerator design flow is augmented to support PLAs. During the creation of the hardware, the datapath is customized for a given loop but is also generalized using the techniques described above. Additional control logic is generated to support the programmable features of the LA. A scheduler-oriented description of the hardware is then generated, containing both information about the datapath as well as the control signals required to direct the datapath. This machine description can then be used by the compiler to map a new loop onto the same hardware.

4.3. Mapping Loops onto a PLA

Conventional modulo schedulers assume a processor with a datapath that is largely homogeneous. For example, FUs are typically ALUs capable of all integer operations, and a centralized register file allows data transfers from any producer FU to any consumer FU. Multicluster VLIWs and CGRAs have more distributed resources, but these architectures are still regular, and scheduling techniques for them exist [17, 19]. Conversely, the loop accelerator datapath contains a significant amount of heterogeneity. FUs have a subset of functionality that is tailored for the loop being accelerated, and connections between FUs are point-to-point and highly irregular. A scheduler targeting an accelerator must accommodate this heterogeneity.

There are various possible approaches to mapping loops onto the PLA. Due to the sparse solution space, traditional operation-centric schedulers (heuristic methods that schedule one operation at a time) are unsuccessful, because they are likely to be trapped in local minima. Simulated annealing [17] is an attractive technique as it can effectively search the large space of possible schedules to find a valid solution. Another approach is to formulate the mapping problem as a set of constraints, and use a constraint solver to find a valid schedule [10]; this is the approach used in this paper. First, MOVs are inserted into the loop's dataflow graph to handle producer-consumer relations that are not in the LA. Then, the assignment of operations to FUs and time slots is formulated as a satisfiability problem and solved. As in conventional modulo scheduling, allocation of rotating registers is performed after assignment of operations to FUs and time slots. If the loop cannot be scheduled at a given II, or if rotating register allocation fails, the II is increased and another scheduling attempt is made. Details about the modulo scheduler can be found in [10].

5. Experimental Results

5.1. Overview

Loop kernels from various signal processing (`fir`, `fft`, `fmradio`, `bfform`), media (`dcac`, `dequant`, `fsed`, `sobel`), and linear algebra (`heat`, `lu`) applications are used to evaluate the efficiency and programmability of the PLA architecture. The loops range in size from 17 operations up to 60 operations. For each loop, the synthesis system is used to generate Verilog corresponding to both single-function and programmable LAs. Synthesis and placement are performed on the Verilog with Synopsys Design Compiler and Physical Compiler using a $0.13\mu m$ standard cell library and a target frequency of 200 MHz. Power analysis is performed using PrimeTime PX after the design has been back-annotated with information about parasitics and switching activity. Three experiments are shown: first, the PLA is compared with single-function LAs as well as with the OR-1200 RISC processor [20], which is a simple, single-issue core with a 5-stage in-order pipeline. This experiment examines the tradeoffs in power efficiency when moving from single-function to semi-programmable to fully programmable hardware. The second

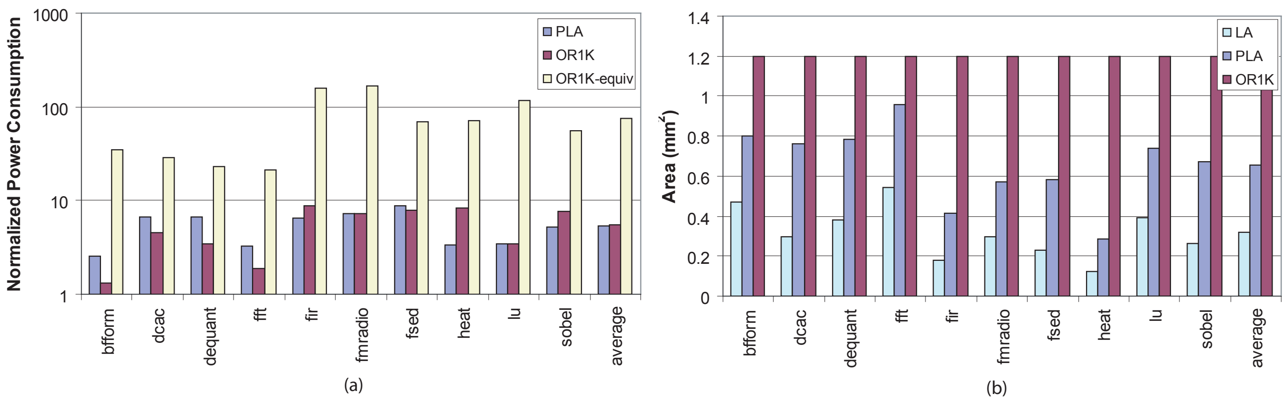


Figure 9: (a) Power consumption of PLA and OR-1200 relative to single-function LA. (b) Area of loop accelerators and OR-1200.

experiment shows the costs of the various PLA datapath generalizations described in Section 4.1. The third experiment measures the programmability of the PLA by mapping loops of varying similarity onto an accelerator.

5.2. PLA Comparison

In the first experiment, the LAs are compared with the OR-1200 processor, which is synthesized at 300 MHz in the same technology ($0.13\mu m$) as the accelerators. The loops are compiled for the processor using a version of the GNU compiler toolchain which has been ported to the OR-1200; optimization level -O2 is used. PrimeTime PX is used to measure the power consumption of the processor given switching activity information obtained during loop execution. Both the local memories in the loop accelerators and the caches in the OR-1200 are included in the power measurements. Figure 9(a) shows the relative power consumption of the single-function LA, the PLA, and the OR-1200 for each loop, on a logarithmic scale. The power consumption of the single-function LA is 1.0; for each loop, the first bar shows the power consumption of the PLA, and the second bar shows that of the OR-1200. In addition, there is a third bar for each benchmark, representing the amount of power the OR-1200 would consume if it ran at a frequency yielding the same performance as the corresponding LA.[3] It is important to note that though the power consumption of the PLAs and the OR-1200 is comparable, the PLAs are 6x to 33x faster than the processor, and this difference in power efficiency is reflected in the performance-equivalent bar.

As the graph shows, the PLAs consume about 2x to 9x more power than the corresponding single-function LAs (which have the same performance). This increased power consumption is due to several factors. First, the power consumed by the RRFs makes up a significant fraction of the overall PLA datapath. When the SRFs in the single-function LA are replaced with RRFs, their sizes must be increased to the next power of two, and additional logic must be added in the form of decoders, adders, and base registers. Also, in the current implementation, the RRFs are synthesized from behavioral descriptions rather than being created by a RF generator, thus missing out on typical RF area and power optimizations such as master latch sharing. The PLA also has other datapath generalizations as described in Section 4.1, such as wider MUXes, which consume additional power. Finally, since the PLA datapath is more complex than the single-function LA, when synthesizing both LAs with the same target clock frequency, the gates in the PLA will be sized larger to meet timing constraints, thus consuming more power.

Comparing the PLA with the OR-1200 at the same performance level, the OR-1200 consumes from 4x to 34x more power. Since the OR-1200 performs general instruction-based execution, it suffers increased power consumption due to factors such as instruction fetch and decode, a centralized register file, caches, and the data forwarding network. Conversely, the PLA is a customized architecture with distributed datapath elements and local memories, and thus is able to achieve high throughput with significantly less power.

Figure 9(b) shows a comparison of the areas of the single-function LA, PLA, and OR-1200. The generalized datapath of the PLA causes its area to increase roughly 2x compared to the single-function LA. Overall, all three hardware implementation styles take up relatively little area, with single-function LAs averaging $0.3mm^2$, PLAs averaging $0.65mm^2$, and the OR-1200 occupying $1.2mm^2$. In terms of area efficiency (performance per area), the PLA is roughly 30x more efficient than the OR-1200 on average for these loops.

Figure 10 plots the performance vs. power consumption of the LAs and OR-1200. On this plot, points on the same slope have roughly equivalent power efficiency in terms of MIPS/mW, with points towards the upper left having greater power efficiency. For each type of hardware, the average (harmonic mean) efficiency is plotted as a line; for the designs studied, the single-function LAs achieve 105 MIPS/mW, the PLAs achieve 24 MIPS/mW, and the OR-1200 achieves 2 MIPS/mW on average.

As can be seen from the plot, the loop accelerators are able to achieve order-of-magnitude improvements in efficiency over the OR-1200 via customization. The PLAs allow hardware reuse in the presence of source code changes, giving up some efficiency to the non-programmable LAs but maintaining large efficiency gains over general purpose hardware. Four commercially available hardware implementations are also shown in the plot: the Tensilica Diamond Core [24], a processor with ASIP-style instruction set extensions optimized for embedded designs; the Texas Instruments C6x digital signal processor [26]; the ARM11 embedded general purpose processor [2]; and the Intel Itanium 2 [13], a general purpose processor tar-

[3]Note that no voltage scaling is done, so the power consumption of the OR-1200 is an underestimate.

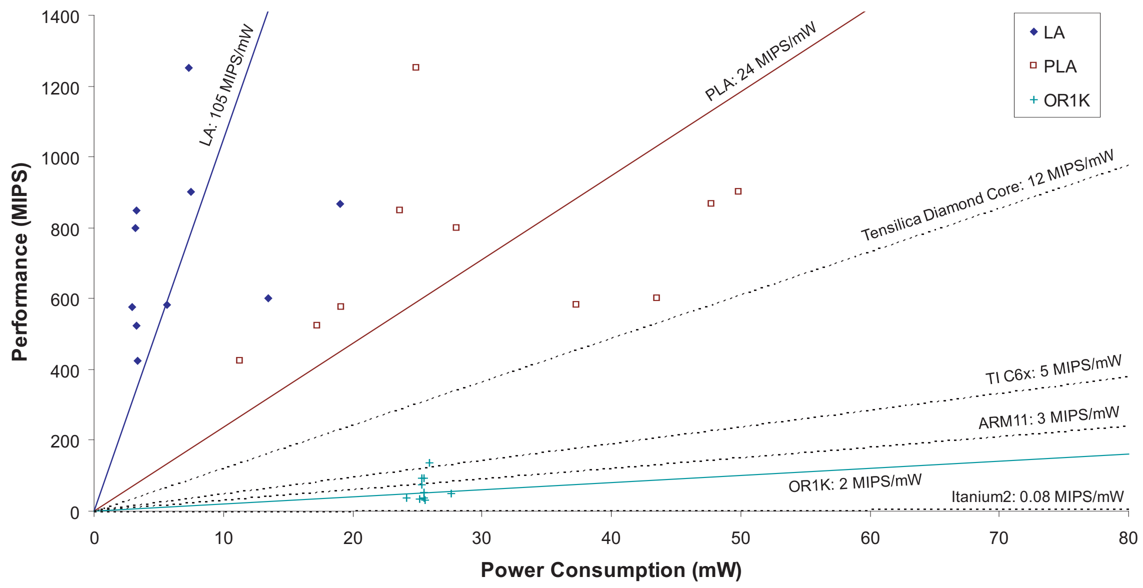


Figure 10: Performance/power of loop accelerators and OR-1200.

geted for enterprise servers. The actual data points for these architectures are outside the range of the plot, but their efficiency lines are shown. As can be observed, the efficiency decreases significantly as the hardware becomes more general and less tailored for embedded applications.

5.3. PLA Datapath Generalizations

Figure 11(a) shows the power overheads of the major datapath generalizations in the PLA. For each loop, a stacked bar shows the breakdown of the amount of power consumption contributed by each datapath element. The power contribution of some datapath elements (such as the global bus) are difficult to isolate when looking at the overall PLA hardware; to measure these contributions, an LA was created which had (for example) a global bus and no other generalizations, and the overall power consumption was compared with that of the original single-function LA. In general, the datapath components contributing the highest amount of overhead are the RRFs and the FU generalizations.

5.4. Programmability

The PLA is meant to be semi-programmable, meaning that once a PLA is designed for a given loop, other loops similar to the given loop should be able to execute on it. In order to measure this programmability, it is useful to be able to systematically generate a series of loops with varying degrees of similarity. To accomplish this, we use the loop perturbation method described in [10], in which random changes (i.e., adding operations, adding or removing edges) are incrementally made to the original loop. After each change, the new loop is increasingly different from the original loop, and the scheduler is less likely to be able to map it onto the PLA without performance loss.

Figure 11(b) shows, for each loop, how many perturbations could be made before the scheduler was no longer able to map the loop onto the hardware without incrementing II. Since the perturbation process is random, multiple runs are performed using different random seeds, and the results are averaged across these runs. Thus, for each loop, the graph shows the number of perturbations for which, on average, II was incremented less than once. A higher number of perturbations indicates that the PLA is more programmable.

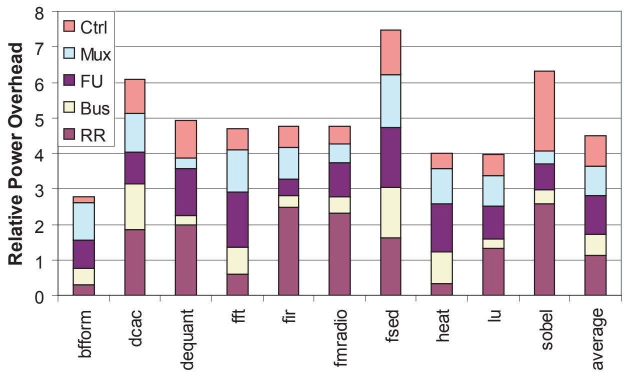
As the graph shows, the programmability of the PLA depends on the original loop. Factors such as more opcodes and more heterogeneous communication patterns in a loop will lead to more programmable hardware. For example, `fir` is a small loop which has simple, repeated communication patterns. Thus, there are fewer unique point-to-point connections in the datapath. On the other hand, `heat` is also a small loop, but its PLA contains more heterogeneous connections.

Note that the graph only shows the number of perturbations possible without performance loss. Generally, even after large numbers of perturbations have been made to a loop, it can still be mapped onto a PLA if some performance loss (increase in II) is tolerable, because the presence of the global bus allows data transfer between any FUs within the PLA.

The PLA is designed to run similar loops (i.e. those obtained via incremental changes to an initial loop) efficiently on the same hardware. Thus, it is unlikely that a given PLA will execute an arbitrary loop efficiently or at all; this is a tradeoff of the PLA design. Nevertheless, we performed a cross-compilation experiment with the 10 loops studied; mapping succeeded in 25 of the 90 cross-compilation opportunities.

6. Conclusion

Customized loop accelerators are able to provide significant performance and power efficiency gains over general purpose processors. By building semi-programmable accelerators, it is possible to achieve these efficiency gains while allowing hardware to be reused as the software evolves. The loop accelerator datapath is generalized in an efficient way such that



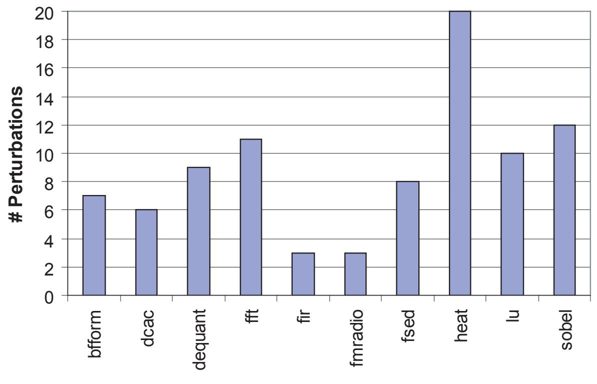


Figure 11: (a) Power consumption breakdown of PLA generalizations. (b) Number of perturbations achieved without loss of performance.

loops that are similar to the original loop may be mapped onto the accelerator. Such programmable loop accelerators provide hardware reusability along with order-of-magnitude improvements in power and area efficiency over simple low power general purpose processors. For the loops in this paper, the PLA was able to achieve 4x-34x better power efficiency and about 30x better area efficiency than a general purpose processor, while losing 2x-9x in power and 2x in area to a custom non-programmable LA.

7. Acknowledgements

We wish to extend our thanks to Shantanu Gupta, Shuguang Feng, and Jason Blome for their help synthesizing the OR-1200 processor. We also thank the anonymous referees for their excellent comments. This research was supported by the National Science Foundation grants CNS-0615261 and CCF-0347411, ARM Ltd., and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] ARC International. Arctangent processor. http://www.arc.com.

[2] ARM. Arm11. http://www.arm.com/products/CPUs/families/ARM11Family.html.

[3] S. Ciricescu et al. The reconfigurable streaming vector processor (RSVP). In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 141–150, 2003.

[4] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, page To appear, June 2008.

[5] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of System Architecture*, 45(1):949–973, 1999.

[6] H. Corporaal and P. Arend. MOVE32INT, a sea of gates realization of a high performance transport triggered architecture. *Microprocessing and Microprogramming*, 38(1):53–60, 1993.

[7] J. Dehnert and R. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1):181–227, May 1993.

[8] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[9] 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.

[10] K. Fan, H. Park, M. Kudlur, and S. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*, pages 124–133, Apr. 2008.

[11] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 324–335, Dec. 1996.

[12] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[13] Intel Corporation, Santa Clara, CA. *Intel IA-64 Software Developer's Manual*, 2002.

[14] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006.

[15] 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.

[16] MAXQ. MAXQ RISC microcontrollers. http://www.maxim-ic.com/products/microcontrollers/maxq/.

[17] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003.

[18] Motorola. *CPU12 Reference Manual*, June 2003. http://e-www.motorola.com/brdata/PDFDB/docs/CPU12RM.pdf.

[19] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 103–114, Dec. 1998.

[20] OpenCores. OpenRISC 1200, 2006. http://www.opencores.org/projects.cgi/web/ or1k/openrisc_1200.

[21] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006.

[22] 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.

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

[24] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf.

[25] Texas Instruments. *TMS320C54X DSP Reference Set*, Mar. 2001. http://www-s.ti.com/sc/psheets/spru131g/spru131g.pdf.

[26] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, July 2006. http://focus.ti.com/lit/ug/spru189g/spru189g.pdf.

[27] M. Woh et al. The next generation challenge for software defined radio. In *Proc. of the 7^{th} International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 343–354, July 2007.