Application-Specific Hardware: Computing Without CPUs

Mihai Budiu
mihaib@cs.cmu.edu

Abstract

In this paper we propose a new architecture for general-purpose computing which combines a reconfigurable-hardware substrate and compiler technology to generate Application-Specific Hardware (ASH). The novelty of this architecture is that resources are not shared: each different static program instruction can have its own dedicated hardware implementation. ASH enables the synthesis of circuits with only local computation structures, which promise to be fast, inexpensive and use very little power. This paper also presents a scalable compiler framework for ASH, which generates hardware from programs written in C and some evaluations of the resources necessary for implementing realistic programs.

1 Introduction

For five decades the relentless pace of technology, expressed as Moore's law, has supplied computer architects with ample materials in the quest for high performance. The abundance of resources has translated into increased complexity[1]. This complexity has already become unmanageable in several respects:

- The verification and testing cost escalates dramatically with each new hardware generation.
- Manufacturing costs (both plant costs and non-recurring engineering costs) have skyrocketed.
- Defect density control becomes very expensive as the feature size shrinks; in the near future we will be unable to manufacture large defect-free integrated circuits.
- The dissipated power density (watts/mm^2) of state-of-the-art microprocessors has already reached values that make air-cooling infeasible [4].
- The clock frequency has increased to a value where global signals across the entire chip are infeasible (the propagation delay exceeds the clock cycle [1]).
- The number of exceptions which require manual interventions generated by the CAD tools grows quickly with design complexity [16].
- Today's processors use extremely complicated hardware structures to enable the exploitation of the instruction-level parallelism (ILP) in large windows; however, the sustained performance is rather low [15].

In Section 2 we propose an alternative approach to implement general-purpose computation, which consists of synthesizing — at compile time — application-specific hardware, on a reconfigurable-hardware substrate. We argue that such hardware can be more efficient than a general-purpose CPU, and can solve or alleviate all of the above problems. We call this model **ASH**, from Application-Specific Hardware.

We propose a way to synthesize directly custom, application-specific dataflow machines in hardware. The ASH machines have low overhead, as they implement the whole application in reconfigurable hardware, and avoid time-multiplexing hardware resources.

The main component of the ASH framework is **CASH**, a Compiler for ASH, presented in Section 3. CASH spans both the realm of traditional compilation and hardware synthesis.

In Section 4 we evaluate the hardware resources needed to implement realistic programs within the ASH model of computation. Section 6 describes some implications of the ASH architecture on computer system design.

2 Application-Specific Hardware

In this section we give an overview of the ASH model of computation. The core of ASH is a reconfigurable fabric; compilation subsumes the role of traditional software compilation and hardware synthesis, translating high-level language programs into hardware configurations.

Reconfigurable hardware devices are hardware devices whose functionality can be changed dynamically (see [10] for a survey). The most common type of device is a Field-Programmable Gate Array and features a set of universal logic gates connected by a switched interconnection network. The logic gates are implemented as look-up tables from small memories; by changing the contents of each memory we change the function computed by each gate. Configuration bits also control the switches on the interconnection network; by choosing which switches are

[1] In this paper we will be mostly concerned with the complexity of microprocessors.

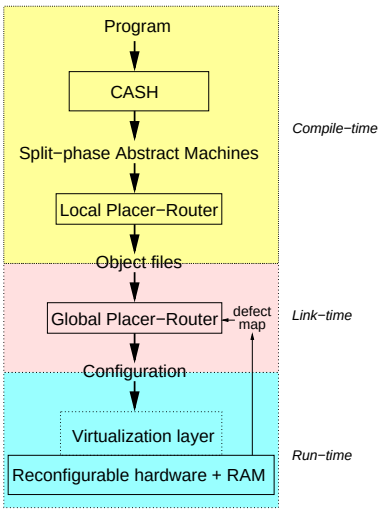

Figure 1: *The ASH tool-flow.*

open and which are closed we effectively connect the logic gates to each other. Reconfigurable hardware thus features the flexibility of general-purpose programmable systems and computation speeds comparable to raw hardware.

Figure 1 summarizes our framework. Programs written in general-purpose high-level languages are the input to the CASH compiler. After applying traditional program-optimization techniques, CASH decomposes the program into small fragments, called Split-phase Abstract Machines, or SAMs.

Each SAM is optimized, synthesized, placed, and routed independently. The placed SAMs that compose the complete program are fed to a global placer and router [2] which decides how to lay-out the machines and how to connect them using an interconnection network. The resulting "executable" is a configuration for the reconfigurable hardware. At run time the configuration is loaded on the reconfigurable-hardware substrate and executed. If the configuration is too large, a run-time hardware-virtualization method may be used.

In this paper we only present the CASH component from Figure 1.

2.1 Split-phase Abstract Machines

The Split-phase Abstract Machine (SAM) is the main abstraction of our intermediate program representation. The compiled program is partitioned into a collection of SAMs, which communicate asynchronously with each other. Each SAM contains computation and possibly a small local memory. The computation implemented in a SAM has predictable latency[3]; moreover, the SAM local memory has predictable access times.

SAMs are inspired by the Threaded Abstract Machine model [11]; like TAMs, whenever a SAM needs to execute an operation that has unpredictable latency it uses the inter-SAM communication network: remote memory accesses, and control-flow transfers between SAMs are transformed into messages routed dynamically on the network. SAMs roughly approximate the procedures in a high-level programming language (however, in our implementation a procedure can be decomposed into several SAMs).

During the program execution, at each instant a SAM can be in one of three states:

- **Inactive SAMs** are not being executed and do not have any live state. These SAMs do not need to consume any power and, if hardware virtualization is available, can be swapped out of the reconfigurable hardware.
- **One active SAM** is actively switching and consuming power and should be entirely swapped in; it is analogous to the procedure on the top of the stack (currently being executed) in a traditional model of computation[4].
- **Passive SAMs** are mostly quiescent: they store live values, but are blocked waiting for the completion of a "callee" SAM. They dissipate only static power most of the time[5] and correspond roughly to the procedures in the current call chain, which have been started in execution, but have not been completed.

2.2 An example

Figure 2 shows a simple C program and the equivalent translation into three SAMs. This figure has been automatically generated by an early version of our prototype CASH compiler using the VCG graph layout tool [18] as a back-end. This figure illustrates just one possible implementation, and a rather suboptimal one.

The compiler creates three SAMs from this program:

SAM 1 implements the initialization of the variables `i` and `j` with 0. It receives as input the "program counter" (PC), which indicates the caller SAM. The shaded empty oval receives a control token which enables the current SAM to start execution. The lightly shaded rectangles with a sharp sign are output registers, containing data that is passed to SAM 2.

[2] The global placer can use a defect map of the target chip to provide fault-tolerance, by avoiding the defective regions.

[3] SAMs can also invoke remote operations, which have unpredictable latencies.

[4] Currently we only consider programs which have a single thread of execution; a parallel model of execution might have several active SAMs at one moment.

[5] There may be some concurrent activity between the passive SAMs and the active one, because of "instructions" that can be executed in parallel with the "call".

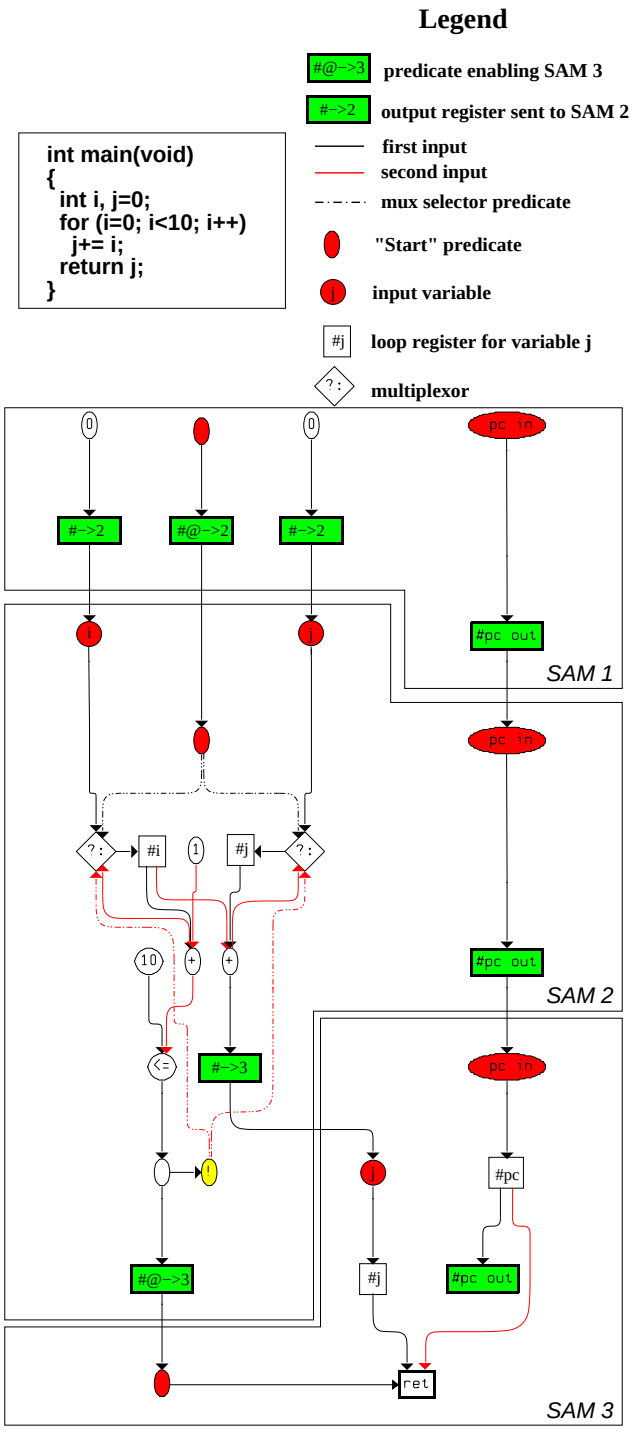

Figure 2: *A simple C program and its equivalent SAM implementation. The tokens are not explicitly represented in this figure: conceptually each wire has an associated token, indicating when the signal on the wire is valid.*

SAM 2 implements most of the computation in the procedure. It contains two additions, one for `i` and one for `j`, a comparison of `i` with 10, and two multiplexors (represented by the `?:` diamonds) that select the values for `i` and `j` based on the flow of control: either the initial value or the result from the increment operation. The multiplexors have two data inputs and two control inputs (dotted lines) each; the shades correspond: when the dark dotted line is asserted, the dark input is selected. The boxes marked with sharp signs # are registers holding the state, represented by the values of `i` and `j`.

SAM 2 is executed as long as the loop condition is true (i.e., `i < 10`). When the loop condition becomes false, control is transfered to the SAM 3.

SAM 3 executes just the `return` instruction. It receives from SAM 2 the value of `j` and the PC and uses them as arguments to the return "instruction". Because the return instruction has a side-effect, it has a third input, a predicate, which indicates when the instruction is safe to execute. Because this return is executed unconditionally, the predicate is fed directly from the enabling token. This return instruction uses the PC value to return the control to the SAM that had originally invoked SAM 1.

2.3 Benefits of the ASH Model

The ASH model has better scalability properties than traditional CPU architectures. For instance:

- The verification and testing of a homogeneous reconfigurable fabric is much simpler. The program is translated directly into hardware, so there's no interpretation layer (i.e., the CPU) which can contain bugs. Moreover, we believe that by building CASH as a certifying compiler [23][6], we can completely eliminate one complex layer needing verification and testing (the processor).
- The manufacturing of reconfigurable circuits reuses the same masks for all circuits, reducing cost.
- As shown by research in the Teramac project [13], reconfigurable hardware architectures can tolerate manufacturing defects through software methods.
- Only the active SAM is switching at any point, requiring very little power.
- The SAM implementation uses only local signals. All inter-SAM communication is made using a switched, pipelined interconnection network. There is no need for global electrical signals.
- CAD tools for reconfigurable hardware can be much simpler than general VLSI tools.
- Dynamic methods of extracting ILP from programs (as implemented in today's out-of-order processors)

[6] A certifying compiler generates not only an executable but also a formal proof that the executable is equivalent with the input program.

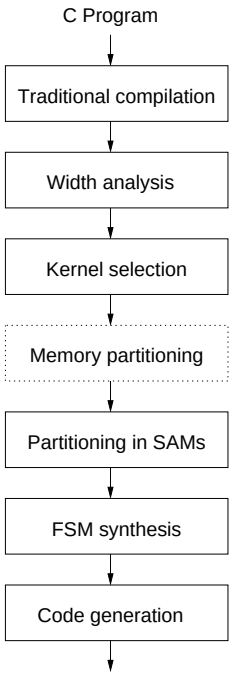

Figure 3: *The CASH compiler passes. The dotted line indicates a component which is not implemented.*

are hindered by limited issue windows: they cannot exploit parallelism outside of a relatively small window of instructions. Our compiler analyzes large program fragments and can uncover substantially more parallelism. We quantify the ILP we discover in Section 4.

The main disadvantage of the ASH paradigm is the requirement for substantial hardware resources. However, this can be alleviated through use of virtualization, or by hardware-software partitioning between a CPU and an ASH fabric. We will quantify the resources necessary in Section 4.

The interaction of the ASH model of computation with the operating system and with multi-tasking is a subject of future research.

3 CASH

In this section we describe CASH, our current implementation of the ASH Compiler. Our compiler infrastructure is built around the SUIF 1.3 research compiler [35]. For the moment, we do not use any of the parallelizing components of SUIF. Figure 3 illustrates the main passes of our compiler. Here are brief descriptions of each of them:

Traditional compilation: most standard compiler front-end optimizations (e.g., dead-code elimination, copy propagation, common subexpression elimination, unreachable code removal) are beneficial in the context of the ASH framework. We also use aggressive procedure inlining.

Width analysis and hardware cost estimation: our reconfigurable hardware target can implement arbitrary-width arithmetic efficiently. We use the BitValue [6] analysis algorithm, which can discover narrow-width scalar operations in C programs. The results presented in this paper do not make use of BitValue, but we plan to incorporate them in future work.

Kernels selection: when we target a system comprised of a CPU and a reconfigurable system, this pass selects program portions that are most likely to provide benefits when executed on the reconfigurable fabric. In the rest of this paper however, we assume that we compile the whole program in reconfigurable hardware, so we do not use any kernel selection algorithm.

Memory partitioning: a variety of techniques can be employed to discover for each piece of code the memory regions that it will access. Once this kind of information is available, various techniques can be used to co-locate the memory and the code accessing it. This part of the compiler is currently not implemented: all memory accesses are made to an external monolithic memory, like in CPU-based systems.

SAM selection: the compiled program is decomposed into split-phase abstract machines. In order to extract a large amount of ILP we implement implement each SAM from one hyperblock [21]. Hyperblocks have been introduced in the context of predicated-code machines, and comprise multiple program basic blocks. We discuss this phase in detail in Section 3.1.

FSM synthesis: from each SAM we generate a finite-state machine (FSM). The FSM has a combinational portion, which computes the next state and a feedback portion, which implements the looping. The FSM state consists of the loop-carried variables. We discuss this compilation phase in detail in Section 3.2.

Code generation: the only back-ends we have implemented so far are a graph-drawing back-end (which was used to generate Figure 2) and a generator which outputs C programs. These programs simulate the execution of the SAMs and compute timing information. In the future, we plan to adapt the back-end to generate HDL descriptions of the SAMs.

3.1 SAM Implementation

In this section we present details about our implementation of the Split-phase Abstract Machines.

The main abstraction we use at the program level is the hyperblock [21]. A hyperblock is a part of the program control-flow graph (CFG) that has a single entry point but possibly multiple exits. The hyperblock may contain loops, but all the loops have to share the same loop entry point, which is also the hyperblock entry point. In our current implementation, each hyperblock becomes a SAM.

Hyperblocks have already been used for generating reconfigurable-hardware implementations in [8, 27]. Our method of hyperblock selection is completely general and deals with unstructured flow of control. We cover each procedure with disjoint hyperblocks, using a linear-time algorithm, as follows:

- A depth-first traversal of the CFG is used to label the back-edges;
- End-points of the back-edges are marked as hyperblock entry points;
- A reverse depth-first traversal is used to assign basic blocks to hyperblocks, as follows:
  - a basic block is in the same hyperblock as all its predecessors;
  - unless two predecessors are from different hyperblocks, in which case the block is itself a hyperblock entry point.

There are several knobs that we can turn to tune the hyperblock selection. We can optimize the resulting circuit for area, speed, or power. One degree of freedom is the traversal order, which defines the back-edges and thus the entry points. A second degree of freedom we have is to add extra entry points, fragmenting large hyperblocks into several small ones. A third degree of freedom is the possibility of duplicating the body of the basic blocks that have multiple hyperblock predecessors and thus creating fewer large hyperblocks (this last technique is used in [21]). We have not explored any of these trade-offs. Our hyperblock selection scheme generalizes all the other proposed schemes[7].

Some hyperblocks will contain loops. Because these loops have exactly one entry point, which coincides with the hyperblock entry point, they are well-structured and the loop-induction variables are well-defined. We can thus synthesize each hyperblock into a finite-state machine.

3.2 FSM Synthesis

We implement each finite-state machine as a dataflow machine [33]. In dataflow machines the computation is expressed as a graph of operations which are triggered by the availability of data. Each data item is encapsulated within a "token", which indicates the functional units that are supposed to process it; when all the inputs of a functional unit are available, the unit consumes the tokens and generates a new one.

In our implementation, the "tokens" are no longer explicitly represented: they become two 1-bit signals connecting the functional units (one bit is used to signal data availability, the other to confirm data consumption). There is no token store, token-matching logic or register file. The main overhead of the interpreted dataflow machines is thus completely eliminated. Static scheduling can eliminate most of the token synchronization.

The computation of the combinational portion of the FSM is implemented speculatively in the style of predicated static-single assignment [9] and predicated speculative execution [2]. To illustrate the implementation, we use the example in Figure 4, a code snippet from the `g721` Mediabench [17] program.

3.3 Path Predicates

Each basic block in a hyperblock has an associated *path predicate*, as described in [9]; the path predicate associated to block B is true if and only if block B is executed during the current loop iteration. The predicates corresponding to blocks are recursively defined:

$$\begin{array}{l} P(entry) = True \\ P(s) = \vee_{p \in Pred(s)} (P(p) \wedge B(p,s)) \end{array} \quad (1)$$

where $B(p,s)$ is true if block p branches to s. This should be read as: "Block s is executed if and only if one of its predecessors p is executed and p branches to s." For the example in Figure 4:

$$\begin{array}{rcl} P(a) & = & True \\ B(a,b) & = & (\texttt{fa1} < -8191) \\ B(a,c) & = & \neg B(a,b) \\ P(b) & = & P(a) \wedge B(a,b) \\ P(c) & = & P(a) \wedge \neg B(a,b) \\ B(c,d) & = & (\texttt{fa1} > 8191) \\ P(d) & = & P(c) \wedge B(c,d) \\ P(e) & = & P(c) \wedge \neg B(c,d) \\ B(b,f) & = & True \\ P(f) & = & (P(b) \wedge B(b,f)) \vee (P(d) \wedge B(d,f)) \\ & & \vee (P(e) \wedge B(e,f)) \end{array}$$

We next use a method equivalent to the *instruction promotion* technique described in [21], which removes predicates from some instructions or replaces them with weaker predicates, enabling their speculative execution. Interestingly enough, if the hyperblock code is in static-single assignment form, we can prove that every instruc-

[7] Other schemes make use of profiling information, which we could easily accommodate.

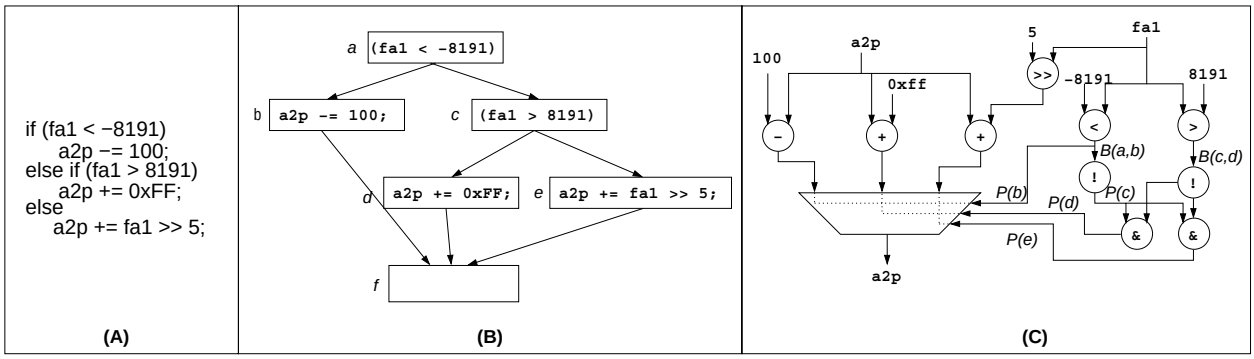

Figure 4: *(A) A code fragment (B) Its control-flow graph (C) Its speculative implementation in hardware.*

tion with no side-effects can be safely and completely promoted to be executed unconditionally.

We will sketch a proof of this fact here. We can distinguish four types of instructions: (a) instructions with side-effects (i.e. memory accesses and procedure calls), (b) predicate computations, (c) multiplexors and (d) all other instructions.

The instructions of type (a) cannot be executed speculatively. We will argue below that instructions of type (b) (i.e. the predicates) can be speculatively computed. It will follow that the multiplexors (c) will always make the correct choices, because they select based on the predicates. It will follow that instructions of type (d) can always be safely executed speculatively, as they have no side-effects and their inputs are always correct.

We now prove that all instructions of type (b), which compute predicate values, can be speculatively executed. The key observation is that, although the predicates are themselves speculatively computed, their value is always correct. We will illustrate the proof using a simple example.

Consider the CFG fragment in Figure 5. According to formula 1, $P(c) = (P(a) \wedge B(a,c)) \vee (P(b) \wedge B(b,c))$. Let us assume that during an execution of the program, basic block a is executed and it branches to c. This means that block b is not executed; thus, the branch condition $B(b,c)$ may have an incorrect value (for instance, because block b changes some values which influence the branch computation). However, by using induction on the depth of the block, we can assume that $P(b)$ has the correct value, False. Thus, the value of $B(b,c)$ is irrelevant for the computation of $P(c)$.

The predicate computation is implemented in hardware, using the same dataflow style. The path predicates are used to guard the execution of instructions with side-effects (memory writes, memory reads that can trigger exceptions, procedure calls and returns). Predicates also control the looping of the FSM; on exit from the current SAM, the predicates also indicate which of the successor

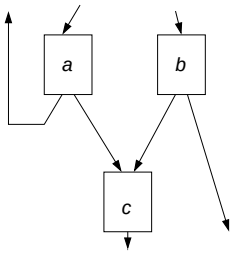

Figure 5: *Fragment of a control-flow graph.*

SAMs should be executed.

This implementation of the program is essentially a static single assignment representation (SSA) [12] of the predicated program. The ϕ functions of the SSA form are essentially multiplexors, selecting among the many definitions of a value that reach a join point in the control-flow graph. Unlike other proposed SSA representations for predicated code [9], we explicitly build the circuitry to compute the ϕ functions, which become multiplexors in hardware (see Figure 4(C)). The multiplexors are controlled by the path predicates. We found that this explicit representation of the complete program is extremely handy, enabling a lot of classical compiler optimizations (dead-code elimination, common-subexpression elimination, constant folding, etc.) to be carried practically in linear time and using very simple and clean algorithms.

The predicates are implemented using formula 1, which implies that their cost is small: each new predicate requires just the logical disjunction of a set of values which were computed previously. Each edge in the CFG contributes one term to the predicate computation, so the implementation of all the predicates together uses resources linear in the size of the hyperblock.

3.4 Eager multiplexors

One problem of the predicated-execution architectures is that the execution time on the speculated control-flow

```
for each basic block (b) in topological order
    L = predecessors(b)
    for each variable (v) live on entry in b
        create a multiplexor M with |L| inputs
        I = 0
        for each block p ∈ L
            V = definer of v at exit of p
            M.input(I) = V
            M.selector(I) = P(p) ∧ B(p, b)
            I = I + 1
        endfor
    endfor
endfor
```

Figure 6: *Algorithm for insertion of multiplexors in a program.*

paths may be unbalanced [3]. For instance, assume that the subtraction takes much longer than the addition; then the leftmost path in Figure 4(C) is the critical path, which dominates the computation time irrespective of which operation should be executed. We have a very simple solution to this problem, which consists of using *eager, fully decoded multiplexors*.

Our multiplexor implementation uses fully-decoded multiplexors, which have as many selector bits as there are inputs. Each selector bit selects one of the inputs, as shown by the dotted lines in Figure 4(C). These multiplexors do not need the complicated encoding/decoding logic for the selection signals and can be very cheaply implemented in hardware, as a wired-or. The eager multiplexor can generate its output as soon as one selector predicate is "True" and the corresponding selected data item is valid[8]

3.5 Multiplexor placement

As we already noted, the placement of multiplexors corresponds to the placement of ϕ functions in SSA form. However, our problem is simpler, because all the back edges in a hyperblock go to the entry point and the rest of the hyperblock can be treated as a feed-forward program fragment. Our current algorithm is presented in Figure 6[9].

We next run a multiplexor simplification pass, which repeatedly uses the following two rules:

[8]Using eager multiplexors might not be enough to guarantee minimal-time execution within a loop. If the speculated paths can be proven statically to be unbalanced we can just avoid executing the long one speculatively or we can use a "reset" signal to abort the computation when it is known that its result is not needed.

[9]For the placement of the multiplexors we plan to implement the algorithm described in [32] which has a much lower complexity. The generated circuit will be the same.

```
procedure merge(m, n)
/* Merge muxes m, n, where m is the k-th input of n */
    remove n's k-th input
    foreach input i of m
        add i as a new j-th input to n
        sel(n, j) = sel(n, k) ∧ sel(m, i)
    endfor
end
```

Figure 7: *Coalescing two chained multiplexors.*

- If a multiplexor has multiple identical inputs, they are merged into a single input and the predicate is set to the logical "or" of the corresponding predicates
- A multiplexor with a single input is removed and the input is connected directly to the output.

Note that the result of this algorithm is not identical to Figure 4(C) (but it is equivalent). However, if we add a third multiplexor simplification rule (see Figure 7), which we have not yet implemented we obtain the same result. This third rule transforms two chained multiplexors (one being the unique output of the other) into a single multiplexor.

We have denoted by $sel(m, i)$ the predicate corresponding to the input i of multiplexor m.

3.6 Tokens

Logically, each data signal has an associated "token" signal, which is used to indicate when the value signal is valid (i.e., the computation that generates the value has terminated). This technique is used in the asynchronous circuit design of micropipelines [30]. The tokens can be implemented as 1-bit wires connecting the producer and consumers of a value; the tokens act as "enable" signals for the consumers. When the computation is inside a loop body, we need also an acknowledgement signal from the consumer to the producer, to indicate consumption of the data value; this indicates when the wires connecting the producer and the consumer can be reused.

We expect that in synchronous hardware implementations most of the token signals can be optimized away by statically scheduling the operations with known latency (more precisely, we can use a single token for all operations executed during the same clock cycle, and we can dispense with the acknowledgement altogether). Passing tokens will remain necessary between producers which have unpredictable latency (i.e., remote operations, like memory reads and procedure calls) and their consumers.

Tokens are used not only to signal that data values are ready, but also to preserve the original program order be-

tween instructions which have side effects. (Most of the previous work on data-flow machines [24, 31, 14] could dispense with this feature because it was handling functional languages.) For instance, two store instructions that have no data dependency between them cannot be reordered if they may update the same memory location.

4 Resources Required for ASH Implementations

In this Section we present a preliminary evaluation of the required resources for the complete implementation of programs in hardware. We analyze a set of programs from the Mediabench [17] and SpecInt95 [28] benchmark suites.

Resources: Table 1 displays the resources required for the complete implementation of these programs in hardware. We do not include in these numbers the standard library or the operating system kernel. We disabled inlining for collecting these numbers. All the values are static counts.

For some of the operations it is fairly easy to estimate the required hardware resources; we listed these under the heading "bits", and the values indicate the approximate number of bit-operations required to implement them. For remote operations (memory access, call/return), the implementation size can vary substantially, depending for instance on the nature of the interconnection network. We report for these just total counts.

The columns in Table 1 are:

LOC: lines of source code, including whitespace and comments, before preprocessing

SAMs: number of SAMs generated

fp: floating-point operations

memory: load and store operations

call/ret: call and return operations

predicates: boolean operations computing predicates (all of them are binary or unary, so each is one operation)

arithmetic: estimated number of operations necessary to implement the integer arithmetic operations (constant multipliers are strength-reduced to a few additions; non-constant multiplies and divisions are assumed to use n^2 bits, where n is the input width)

mux: number of bit operations in multiplexors (number of inputs * input size)

loop_regs: number of bits in loop registers

Comments: The raw computation resources required (the total of the "bits" columns) is below 2 million for all benchmarks except `mesa`, which is below 5 million. Even by today's standards, these are reasonably small (modern CPU cores already use more than 30 million transistors each, and state-of-the art reconfigurable fabrics already provide roughly 1-million bit-operations). By discounting the density disadvantage for the reconfigurable circuits, but extrapolating using Moore's law, within the next 10 years we have enough resources to implement each of these programs completely in hardware.

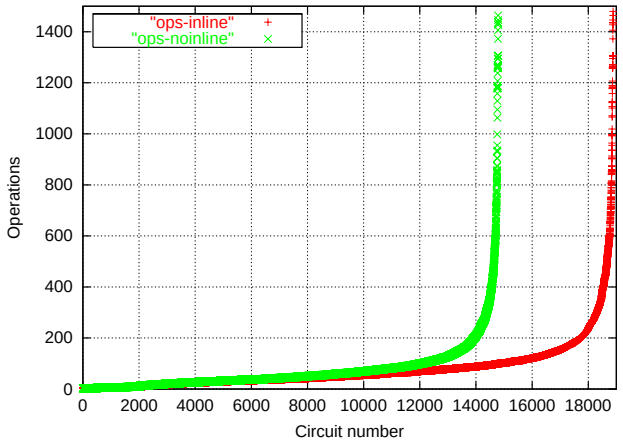

Figure 8: *Operations in each SAM, with and without procedure inlining. The data is for all the SAMs synthesized from all our benchmarks. The SAMs are sorted on their size and numbered in increasing order. On the x axis we have the SAM number and on the y axis the total number of "operations" per SAM. About 20 outliers above 1500 have been left out of the picture: the maximum size is 13100 without inlining and 14000 with inlining.*

This data doesn't include the savings that can be achieved by implementing computations of custom sizes. Research has shown [20, 6, 29] that static methods can eliminate a lot of the manipulated bits (methods developed by our own research [6] indicate that 20% of the bit computations in these benchmarks can be eliminated).

Notice that the resources taken by the predicate computations are minor compared to the actual computation; this suggests that large-granularity reconfigurable fabrics are more suitable for ASH systems than today's dominant style of FPGA, which has 1-bit functional units.

SAM/hyperblock size: Figure 8 shows a distribution of the SAM sizes, when measured in operations (this corresponds to the hyperblock size). If we use procedure inlining, we obtain more circuits, which tend to be larger. 90% of the SAMs use less than 200 operations.

Comments: Most SAMs are relatively small, which will translate into reduced power consumption and good locality for the intra-SAM signals.

ILP: In Figure 9 we plot the "average instruction-level

Benchmark	LOC	SAMs	Units			Bit-operations			
			fp	memory	call/ret	predicates	arithmetic	mux	loop_regs
adpcm_e	302	8	0	19	9	51	8,128	3,014	646
adpcm_d	302	8	0	19	9	51	6,144	3,014	646
g721_Q_e	1613	43	0	177	138	483	42,883	9,032	1,766
g721_Q_d	1619	41	0	180	137	486	42,837	8,804	1,635
gsm_e	6074	218	0	1,780	517	1,942	413,794	38,694	9,871
gsm_d	6070	214	0	1,768	513	1,936	413,014	38,436	9,805
epic_e	2701	312	75	498	295	1,335	300,665	69,848	29,689
epic_d	2452	231	36	719	242	1,125	230,418	66,922	28,021
mpeg2_e	7605	366	197	2,724	877	3,275	537,906	85,504	23,436
mpeg2_d	9832	316	11	1,789	839	2,624	305,803	45,570	13,333
jpeg_e	26881	1,331	153	8,693	1,964	8,167	1,022,023	200,354	76,722
jpeg_d	26115	1,285	153	8,248	1,889	7,625	996,243	194,374	75,897
pegwit_e	6713	270	0	2,112	608	1,346	151,160	24,039	7,857
pegwit_d	6713	270	0	2,112	608	1,346	151,160	24,039	7,857
mesa	65806	3,165	6,269	24,779	7,301	38,779	3,170,972	709,566	258,516
129.compress	1934	71	4	268	97	285	37,056	7,452	2,804
099.go	29246	1,778	0	9,610	2,966	19,350	1,309,148	367,116	104,987
130.li	7608	616	13	2,321	1,675	3,106	180,823	51,996	9,884
132.ijpeg	29265	1,427	163	8,556	2,321	8,141	1,101,735	202,652	78,630
134.perl	27072	1,406	48	14,348	4,997	34,363	1,377,612	467,864	99,015
147.vortex	67210	1,433	4	24,913	9,602	39,195	1,448,933	239,839	32,850

Table 1: *Static resource consumption for each benchmark. Some resources are expressed in units, while other are expressed in bit-operations.*

parallelism" in each SAM. We obtain this value by dividing the number of operations in a SAM (excluding inputs, outputs and constants) by the longest path within the SAM. The "average ILP" does not necessarily correspond to the dynamic ILP; the two values would be identical if all operations would execute with a latency of 1 clock cycle.

Comments: There are a few anomalous SAMs, which have an ILP of 0 or less than 1; one such SAM is SAM 1 in Figure 2 which contains no computations, so has a 0 count of useful operations. These SAMs can be eliminated by a special constant-propagation pass which we have not implemented yet.

We have isolated the ILP available just in the circuits that contain at least one feedback loop. These correspond directly to the program innermost loops, at least for the case of structured flow of control[10].

Fortunately, the ILP for the SAMs which contain loops is quite high: when we use inlining, 90% of the loops have an ILP above 2, about 50% have an ILP above 3, while 20% have an ILP above 5! In a sense (modulo the assumption about the identical latency of all operations), this ILP is the *sustained* ILP of these SAMs. These numbers enable us to believe that the performance of this model of computation can match or even exceed the performance

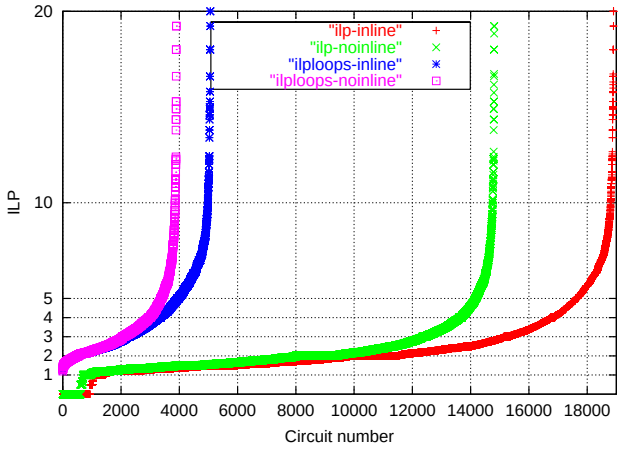

Figure 9: *Average ILP for each SAM, with and without inlining, and considering all SAMs or just the SAMs with loops. The data is for all the SAMs synthesized from all our benchmarks. The SAMs are sorted according to their ILP and numbered. The x axis is the SAM number. There are a few outliers above 20 not shown here: 4 SAMs have an ILP above 20, with a maximum of 41.55.*

[10]Note that outer loops in the program map to several SAMs, so a SAM can be executed within a loop even if it contains no looping inside.

of CPU-based systems.

Notice that we are being exceedingly conservative in scheduling the operations with side-effects: all memory operations and call/returns are enforced (using token-passing) to be made in the original program order, even if they are independent. We even impose a total ordering between memory reads! Once we remove this restriction, the ILP will definitely grow further. Notice that we have implemented common-subexpression and dead-code elimination, so the ILP is not artificially inflated by sub-optimal circuits.

5 Related Work

This work has two different lineages: research on intermediate program representation and compilation for reconfigurable hardware architectures.

Several researchers have addressed the problem of compiling high-level languages for reconfigurable and non-traditional architectures: [5, 22, 19, 7, 27, 26]. To our knowledge, our approach is unique in that it compiles very complex applications to hardware and it doesn't use a fixed number of computational resources.

The output of our compiler is a series of circuits. These bear a striking resemblance to some forms of intermediate representations of the program in other optimizing compilers. Our circuits are closely related to static-single assignment [12], dependence flow graphs [25] and value-dependence graphs [34], and, most closely, to gated-single assignment [32].

However, our circuits explicitly use predication and the notion of hyperblock; in that direction we are indebted to compilation for predicated execution machines: [21] and predicated static single assignment [9]. Unlike these program representations, we explicitly build the code to compute the predicates and we instantiate the ϕ functions through the use of multiplexors.

Our circuits are closely related to dataflow machines (see [33] for a survey), but our circuits are meant to be implemented directly in hardware and not interpreted on a dataflow machines using token-passing. The notion of Split-phase Abstract Machine is derived from the Threaded-Abstract Machine [11], a derivative of the dataflow work.

6 Conclusions

In this paper we have presented a proposal for a new model of computation, called Application-Specific Hardware (ASH), which implements programs completely in hardware, on top of a reconfigurable hardware platform. Our preliminary evaluations enable us to believe that soon we will have enough hardware resources to accommodate complete realistic programs, and that the sustained performance of this model will be comparable to processor-based computations.

We have discussed the compilation technology which can scalably translate large programs written in high-level languages into hardware implementations. Our compilation strategy transforms hyperblocks into circuits which execute many operations speculatively, and thus expose a substantial amount of instruction-level parallelism.

We have also outlined those features of the ASH model of computation that promise to provide *scalability* to this model: ASH implementations can easily and naturally take advantage of the exponentially increasing amount of hardware resources, avoiding many of the problems that the increased complexity brings to standard CMOS-based microprocessor design and manufacturing.

References

[1] V. Agarwal, H.S. Murukkathampoondi, S.W. Keckler, and D.C. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[4] Kaveh Azar. The History of Power Dissipation. *Electronics Cooling Magazine*, 6 (1), 2000.

[5] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing Applications into Silicon. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[6] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[7] Timothy J. Callahan and John Wawrzynek. Instruction Level Parallelism for Reconfigurable Computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinn, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

[8] Timothy J. Callahan and John Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *Pro-*

ceedings International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2000, 2000.

[9] L. Carter, E .Simon, B. Calder, L. Carter, and J. Ferrante. Path Analysis and Renaming for Predicated Instruction Scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[10] Katherine Compton and Scott Hauck. Configurable Computing: A Survey of Systems and Software. Technical report, Northwestern University, Dept. of ECE, 1999.

[11] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, July 1993.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[13] James R. Heath, Philip J. Kuekes, Gregory S. Snider, and R. Stanley Williams. A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. *Science*, 280, 1998.

[14] Steven K. Heller. Efficient Lazy Data-Structures on a Dataflow Machine. Technical Report MIT-LCS-TR-438, Massachusetts Institute of Technology, 1989.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, second edition*. Morgan Kaufmann, 1996.

[16] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[18] I. Lemke and G. Sander. Visualization of Compiler Graphs. Technical Report Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, 1993.

[19] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *DAC 2000*, 2000.

[20] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Sensitive Code Generation in a Custom Embedded Accelerator Design System. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2001)*, St. Goar, Germany, March 2001.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.

[22] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceeding of the International Conference on Computer Architecture 2000*, June 2000.

[23] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, 2000.

[24] Gregory Michael Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[25] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In SIGPLAN, editor, *In Principles of Programming Languages*, volume Volume 18, 1991.

[26] Rahul Razdan. *PRISC: Programmable reduced instruction set computers*. PhD thesis, Harvard University, May 1994.

[27] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A Technology-Scalable Architecture for Fast Clocks and High ILP. In *5th Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[28] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[29] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000.

[30] Ivan Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6)(720–738), June 1989.

[31] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report MIT-LCS-TR-370, MIT, August 1986.

[32] Peng Tu and David Padua. Efficient Building and Placing of Gating Functions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 47 – 55, 1995.

[33] Arthur H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.

[34] D. Weise, R. F. Crew, M. Ernst, and B Steensgaard. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the Twentyfirst Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 297–310, January, 1994.

[35] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.