Spatial Computation

Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea and Seth Copen Goldstein
{mihaib,girish,tibi,seth}@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

This paper describes a computer architecture, *Spatial Computation* (SC), which is based on the translation of high-level language programs directly into hardware structures. SC program implementations are completely distributed, with no centralized control. SC circuits are optimized for *wires* at the expense of computation units.

In this paper we investigate a particular implementation of SC: ASH (Application-Specific Hardware). Under the assumption that computation is cheaper than communication, ASH replicates computation units to simplify interconnect, building a system which uses very simple, completely dedicated communication channels. As a consequence, communication on the datapath never requires arbitration; the only arbitration required is for accessing memory. ASH relies on very simple hardware primitives, using no associative structures, no multiported register files, no scheduling logic, no broadcast, and no clocks. As a consequence, ASH hardware is fast and extremely power efficient.

In this work we demonstrate three features of ASH: (1) that such architectures can be built by automatic compilation of C programs; (2) that distributed computation is in some respects fundamentally different from monolithic superscalar processors; and (3) that ASIC implementations of ASH use three orders of magnitude less energy compared to high-end superscalar processors, while being on average only 33% slower in performance (3.5x worst-case).

Categories and Subject Descriptors: B.2.4 arithmetic and logic cost/performance, B.6.3 automatic synthesis, optimization, simulation B.7.1 algorithms implemented in hardware, B.7.2 simulation, C.1.3 dataflow architectures, hybrid systems, D.3.2 data-flow languages, D.3.4 code generation, compilers, optimization

General Terms: Measurement, Performance, Design.

Keywords: spatial computation, dataflow machine, application-specific hardware, low-power.

1. INTRODUCTION

The von Neumann computer architecture [108] has proven to be extremely resilient despite numerous perceived shortcomings [7]. Computer architects have continuously enhanced the structure of the central processing unit, taking advantage of Moore's law. Today's superpipelined, superscalar, out-of-order microprocessors are amazing achievements.

However, the future scalability of superscalar (and even VLIW) architectures is questionable. Attempting to increase the pipeline width of the processor beyond the current four or five instructions per cycle is difficult since the interconnection networks scale superlinearly. The register file, instruction issue logic, and pipeline forwarding networks grow quadratically with issue width, making the interconnection latency the limiting factor [2]. This problem is compounded by the increasing clock rates and shrinking technologies: currently signal propagation delays on inter-module wires dominate logic delays [56]. Just the distribution of the clock signal is a major undertaking [10].

Wire delays are not the only factor in the way of scaling: power consumption and power density have reached dangerous levels, due to increased amounts of speculative execution, increased logic density and wide issue. Design complexity is yet another limitation: while the number of available transistors grows by 58% annually, designer productivity only grows by 21% [1]. This exponentially increasing productivity gap has been historically covered by employing larger and larger design and verification teams, but human resources are economically hard to scale.

The research presented in this paper is aimed directly at these problems. We explore Spatial Computation, which is a model of computation optimized for wires. We have previously proposed to use Spatial Computation for mapping programs to nanoFabrics [51]; in this paper we evaluate the compiler technology we developed for nanoFabrics on a traditional CMOS substrate. Since the class of circuits one could call "spatial" is arguably very large, we focus our attention on a particular set of instances of SC structures, which we call Application-Specific Hardware (ASH). ASH requires no clocks, nor any global signals. The core assumption is that computation gates are cheap, and will become even cheaper compared to the cost of wires (in terms of delay, power and area). ASH is an extreme point in the space of SC architectures: in ASH computation structures are never shared, and each program operation is synthesized as a different functional unit.

We present a complete compiler/CAD tool-chain that bridges both software compilation and microarchitecture. Applications written in high-level languages are compiled into hardware descriptions. These descriptions can either be loaded onto a reconfigurable hardware fabric or synthesized directly into circuits. The resulting circuits use only localized communication, require neither broadcast nor global control, and are self-synchronized. The compiler we have developed is automatic, fast, requires no designer intervention, and exploits instruction-level parallelism (ILP) and pipelining.

The novel research contributions described in this paper are:
(1) a compiler tool-chain from ANSI C to asynchronous hardware;

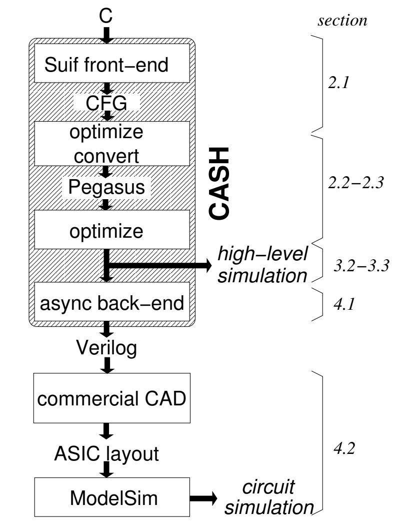

Figure 1: *Toolflow used for evaluation. The right-hand side indicates the sections in this paper that discuss each of the building-blocks.*

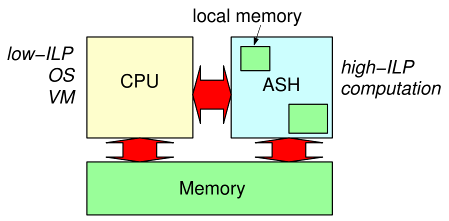

Figure 2: *ASH used in tandem with a processor for implementing whole applications. The processor is relegated to low-ILP program fragments and for executing the operating system.*

(2) a qualitative comparison of Spatial Computation architectures and superscalar processors;
(3) a circuit-level evaluation of the synthesized circuits on C program kernels from the Mediabench suite;
(4) a description of a high-level synthesis toolflow that produces extremely energy-efficient program implementations: comparable with custom hand-tuned hardware designs, and three orders of magnitude better than superscalar processors;
(5) the first implementation of a C compiler that can target dataflow machines.

2. COMPILING C TO HARDWARE

This section presents our compilation methods as embodied in the CASH compiler (Compiler for ASH). The structure of CASH, and its place within a complete synthesis tool-flow are illustrated in Figure 1, which also shows the organization of this paper.

The circuits generated by CASH cannot handle system calls. For translating whole application we assume first that hardware-software partitioning is performed, and that part of the application is executed on a traditional processor (e.g., I/O), while the rest is mapped to hardware, as shown in Figure 2. The processor and the hardware have access to the same global memory space, and there is some mechanism to maintain a coherent view of memory. Crossing the hardware-software interface can be hidden by employing a stub compiler, which encapsulates the information transmitted across the interface, as we have proposed in [20], effectively performing Remote Procedure Calls across the hw/sw interface.

2.1 CASH

CASH takes ANSI C as input. CASH represents the input program using Pegasus [16, 17], a dataflow intermediate representation (IR). The output of CASH is a hardware dataflow machine which directly executes the input program. Currently CASH generates a structural Verilog description of the circuits.

CASH has a C front-end, based on the Suif 1 compiler [111]. The front-end performs some optimizations (including procedure inlining, loop unrolling, call-graph computation, and basic control-flow optimizations), intraprocedural pointer analysis, and live-variable analysis. Then, the front-end translates the low-Suif intermediate representation into Pegasus. Next CASH performs a wealth of optimizations on this representation, including scalar-, memory- and Boolean optimizations. Finally, a back-end performs peephole optimizations and generates code.

The translation of C into hardware is eased by maintaining the same memory layout of all program data structures as implemented in a classical CPU-based system (the heap structure is practically identical, but CASH uses less stack space, since it never needs to spill registers). ASH currently uses a single monolithic memory for this purpose (see Section 4.1.3). There is nothing intrinsic in Spatial Computation that mandates the use of a monolithic memory; on the contrary, using several independent memories (as suggested for example in [94, 9]) would be very beneficial.

2.2 The Pegasus Intermediate Representation

The key technique allowing us to bridge the semantic gap between imperative languages and asynchronous dataflow is Static Single Assignment (SSA) [34]. SSA is an IR used for imperative programs in which each variable is assigned to only once. As such, it can be seen as a functional program [5]. Pegasus represents the scalar part of the computation of C programs as SSA. Due to space limitations we only briefly describe Pegasus. See [16, 17] for more details.

Pegasus seamlessly extends SSA—representing memory dependences, predication, and (forward) speculation in a unified manner. While other IRs have previously combined some of these aspects, we believe Pegasus is the first to unify them into a coherent, semantically precise representation.

A program is represented by a directed graph in which nodes are operations and edges indicate value flow; an example is shown in Figure 3. Pegasus leverages techniques used in compilers for predicated execution machines [74] by collecting multiple basic blocks into one hyperblock;[1] each hyperblock is transformed into straight-line code through the use of predication, using techniques similar to PSSA [25]. Instead of SSA ϕ-nodes, within hyperblocks Pegasus uses explicit decoded multiplexor (MUX) nodes (one example is given in Figure 7). A decoded MUX has n data inputs and n predicates. The data inputs are the reaching definitions. The MUX predicates correspond to path predicates in PSSA; each predicate selects one corresponding data input. The predicates of each MUX are guaranteed to be mutually disjoint (i.e., the predicates are *one-hot encoded*). CASH uses the `espresso` [13] Boolean optimizer to simplify the predicate computations.

Speculation is introduced by predicate promotion [73]: the predicates that guard instructions without side-effects are weakened to become *true*, i.e., these instructions are executed unconditionally once a hyperblock is entered. Predication and speculation are thus core constructs in Pegasus. The former is used for translating control-flow constructs into dataflow; the latter for reducing the crit-

[1]A hyperblock is a portion of the program control-flow graph having a single entry point and possibly multiple exits.

```
int squares()
{
  int i = 0,
      sum = 0;

  for (;i<10;i++)
      sum += i*i;
  return sum;
}
```

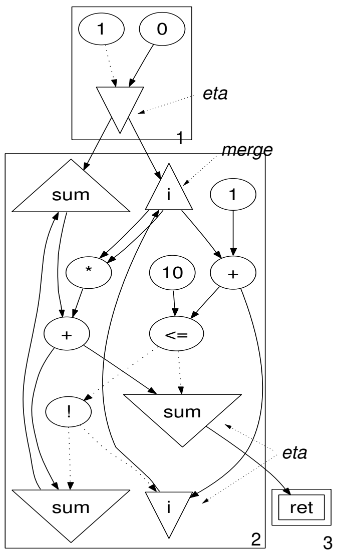

Figure 3: *C program and its representation comprising three hyperblocks; each hyperblock is shown as a numbered rectangle. The dotted lines represent predicate values. (This figure omits the token edges used for memory synchronization.)*

icality of control-dependences [63]. They effectively increase the exposed ILP. Note that MUX nodes are natural speculation squashing points, discarding all of their data inputs corresponding to *false* predicates (i.e., computed on mis-speculated paths).

Hyperblocks are stitched together into a dataflow graph representing an entire procedure by creating dataflow edges connecting each hyperblock to its successors. Each variable live at the end of a hyperblock is forwarded through an ETA node [82] (also called a "gateway"). ETAs are shown as triangles pointing down in our figures. ETA nodes have two inputs—a value and a predicate—and one output. When the predicate evaluates to *true*, the ETA node moves the input value to the output; when the predicate evaluates to *false*, the input value and the predicate are simply consumed, generating no output. A hyperblock with multiple predecessors receives control from one of several different points; inter-hyperblock join points are represented by MERGE nodes, shown as triangles pointing up.

Figure 3 shows a function that uses `i` as an induction variable and `sum` to accumulate the sum of the squares of `i`. On the right is the program's Pegasus representation, which consists of three hyperblocks. Hyperblock 1 initializes `sum` and `i` to 0. Hyperblock 2 represents the loop; it contains two MERGE nodes, one for each of the loop-carried values, `sum` and `i`. Hyperblock 3 is the function epilog, containing just the RETURN. Back-edges within a hyperblock denote loop-carried values; in this example there are two such edges in hyperblock 2; back-edges always connect an ETA to a MERGE node.

Memory accesses are represented through explicit LOAD and STORE nodes. These and other operations with side-effects (e.g., CALL and DIVISION—which may generate exceptions) also have a predicate input: if the predicate is *false*, the operation is not executed. In our figures, predicate values are shown as dotted lines.

The compiler adds dependence edges, called *token edges*, to explicitly synchronize operations whose side-effects may not commute. Operations with *memory side-effects* (LOAD, STORE, CALL, and RETURN) all have a token input. Token edges explicitly encode data flow through memory. An operation with memory side-effects must collect tokens from all its potentially conflicting predecessors (e.g., a STORE following a set of LOADs). The COMBINE operator is used for this purpose. COMBINE has multiple token inputs and a single token output; it generates an output after it receives all its inputs. It has been noted for the Value Dependence Graph representation [97] that such token networks can be interpreted as SSA for memory, where the COMBINE operator corresponds to a ϕ-function. Tokens encode both true-, output- and anti-dependences, and are "may" dependences. We have devised new algorithms for removing redundant memory accesses which exploit predicates and token edges in concert [18, 19]. As we show later, tokens are also explicitly synthesized as hardware signals, so they are *both compile-time and run-time* constructs.

Currently the compiler is purely static, i.e., uses no profiling information. There is no reason that profiling cannot be incorporated in our tool-chain. Section 4.1.2 explains why profiling is less critical for CASH than for traditional ILP compilers [92].

2.3 The Dataflow Semantics of Pegasus

In [16] we have given a precise and concise operational semantics for all Pegasus constructs. At run-time each edge of the graph either holds a value or is $\perp$ ("empty"). An operation begins computing once all of its required inputs are available. It latches the newly computed value when its output is $\perp$. The computation consumes the input values (setting the input edges to $\perp$) and produces an output value. This semantics is the one of a static dataflow machine (i.e., each edge can hold a single value at one time).

The precise semantics is useful for reasoning about the correctness of compiler optimizations and is a precise specification for compiler back-ends. Currently CASH has three back-ends: (1) a graph-drawing back-end, which generates drawings in the `dot` language [45], such as in Figure 3; (2) a simulation back-end, which generates an interpreter of the graph structure (used for the analysis in Section 3); and (3) the asynchronous circuits Verilog back-end, described in Section 4.1 (used for the evaluation in Section 4).

2.4 Compiler Status

The core of CASH handles all of ANSI C except `longjmp`, `alloca`, and functions with a variable number of arguments. While the latter two constructs are relatively easy to integrate, handling `longjmp` is substantially more difficult. Strictly speaking, C does not have exceptions [61] p. 200, and our compiler does not handle them. Recursion is handled in exactly the same way as in software: CASH allocates stack frames for saving and restoring the live local variables around the recursive call. As an optimization, CASH uses the call-graph to detect possibly recursive calls, and avoids saving locals for all non-recursive calls.

The asynchronous back-end is newer, and, therefore, somewhat less complete: it does not yet handle procedure calls and floating-point computations. The latter can be easily handled with a suitable IP core containing implementations of floating-point arithmetic. Currently we handle some procedure calls by inlining. Handling function pointers requires an on-chip network, as the call will need to dynamically route the procedure arguments to the callee circuit dependent on the run-time value of the pointer.

3. ASH VERSUS SUPERSCALAR

This section is devoted to a comparison of the properties of ASH and superscalar processors. The comparison is performed by executing whole programs using timing-accurate simulators. Since there are many parameters, one should see this comparison as a limit study. This study can also be interpreted as being the first head-to-head comparison between an unlimited-resource static dataflow machine and a superscalar processor. Interestingly enough, despite the limited resources of the superscalar, some of its capabilities give it a substantial edge over the static dataflow model, as shown below. These results may be a partial explanation of the demise of the dataflow model of computation, which was a very popular research subject in the seventies and eighties. But first we will briefly discuss the main source of parallelism in dataflow machines.

3.1 Dataflow Software Pipelining

A consequence of the dataflow nature of ASH is the automatic exploitation of pipeline parallelism. This phenomenon has been studied extensively in the dataflow literature under names such as dataflow software pipelining [46], and loop unraveling [33]. As the name suggests, this phenomenon is closely related to software pipelining [3], which is a compiler scheduling algorithm used in mostly for VLIW processors.

The program in Figure 3 illustrates this phenomenon. Let us assume that the multiplier implementation is pipelined with five stages. In Figure 4 we show a few consecutive snapshots of this circuit as it executes, starting with the initial snapshot in which the two MERGEs contain the initial values of `i` and `sum`. (We have implemented a tool that can automatically generate such pictures; the inputs to the tool are pictures generated by the CASH back-end and execution traces generated by the execution simulator.) In the last snapshot (6), the computation if `i` has already executed two iterations, two consecutive values of `i` are injected in the multiplier, while the computation of `sum` has yet to complete its first iteration. The execution of the multiplier is thus effectively pipelined.

A similar effect can be achieved in a statically scheduled computation by explicitly software pipelining the loop, scheduling the computation of `i` to occur one iteration ahead of `sum`. Pipelining also occurs automatically (i.e., without any compiler intervention) in superscalar processors if there are enough resources to simultaneously process instructions from multiple instances of the loop body. In practice large loops may not be dynamically pipelined by a superscalar due to in-order instruction fetch, which can prevent some iterations from getting ahead.

Maximizing the throughput of a pipelined computation in ASH requires that the delay of all paths between different strongly connected components in the Pegasus graph be equal. CASH inserts FIFO elements to achieve this, a transformation closely related to "pipeline balancing" in static dataflow machines [46] and "slack matching" in asynchronous circuits [70]. The FIFO elements correspond to the reservation stations in superscalar designs, and to the rotating registers in software pipelining.

3.2 ASH Versus Superscalar

For comparing ASH with a superscalar we make the following assumptions: (1) all arithmetic operations have the same latencies on both computational fabrics; (2) MUXs, MERGEs and Boolean operations in ASH have latencies proportional to the $\log$ of the number of inputs; (3) ETA has the same latency as an addition; (4) memory operations in ASH incur an additional cost for network arbitration compared to the superscalar; (5) the memory hierarchy used for both models is identical: an LSQ and a two-level cache hierarchy[2].

The superscalar is a 4-way out-of-order SimpleScalar simulation [21] with the PISA instruction set, using `gcc -O2` 2.7.2 as a compiler. ASH is simulated using a high-level simulator which is automatically generated by CASH, as shown in Figure 1. We cannot simulate the execution of libraries in ASH (unless we supply them to the compiler as source-code), and thus we have instrumented SimpleScalar to ignore their execution time, in order to have a fair comparisons.

[2]For this study we use a very similar LSQ for both ASH and the superscalar. As future work we are exploring the synthesis of program-specific LSQ structures.

Naively one would expect ASH to execute programs strictly faster than the superscalar (assuming comparable compiler technology) since it benefits from (a) unlimited parallelism, (b) no resource constraints, (c) no instruction fetch/decode/dispatch, and (d) dynamic scheduling.

Simulating whole programs from SpecInt95 under these assumptions results in two programs (`099.go` and `132.ijpeg`) showing a 25% improvement on ASH, while the other programs are between 10% and 40% slower. The speed-ups on ASH are attributable to the increased ILP due to the unlimited number of functional units; (for these benchmarks the instruction cache of the processor did not seem to be a bottleneck). In the next section we investigate the slowdowns.

3.3 Superscalar Advantages

In order to understand the advantages of the superscalar processor we have carried out a detailed analysis of code fragments which perform especially poorly on ASH. The main tool we have used for this purpose is the *dynamic critical path* [43]. In ASH the dynamic critical path is a sequence of *last-arrival events*. An event is "last-arrival" if it is the one that enables the computation of a node to proceed. Events correspond to signal transitions on the graph edges. The dynamic critical path is computed by tracing the edges corresponding to last-arrival events backwards from the last operation executed. Most often a last-arrival edge is the last input arriving at an operation. However, for lenient operations (see Section 4.1.2), the last-arrival edge is the edge enabling the computation of the output. Sometimes all the inputs may be present but an operation may be unable to compute because it has not received the *acknowledgment* signal for its previous computation; in this case the ack is the last-arrival event.

Despite the fact that the superscalar has to time-multiplex a small number of the computational units, some of the mechanisms it employs provide clear performance advantages. Below is a brief summary of our findings.

Branch prediction: the ability of a superscalar to predict branch outcomes changes radically the structure of the dynamic dependences: for example, a correctly predicted branch is *dynamically independent* of the actual branch condition computation. Unless the in-order commit stage (or some other structural hazard of the processor pipeline) is a bottleneck, the entire computation of the branch condition is removed from the critical path.

In contrast, in ASH inter-hyperblock control transfers are never speculative. Often, the ETA control predicate computation is on the critical path; e.g., when there is no computation to overlap with the branch condition evaluation, such as in "control-intensive" code. Such code fragments may be executed faster on a processor.

Some branches, such as those testing exceptional conditions (e.g., introduced by the use of `assert` statements), are never executed, and thus the processor branch prediction does a very good job of handling them. These cases are especially detrimental to ASH.

We note that good branch prediction requires "global" information, aggregating information from multiple branches, and would be very challenging to implement efficiently in Spatial Computation.

Synchronization: MERGE and MUX operations have a non-zero cost, and may translate in overhead in ASH. These operations cor-

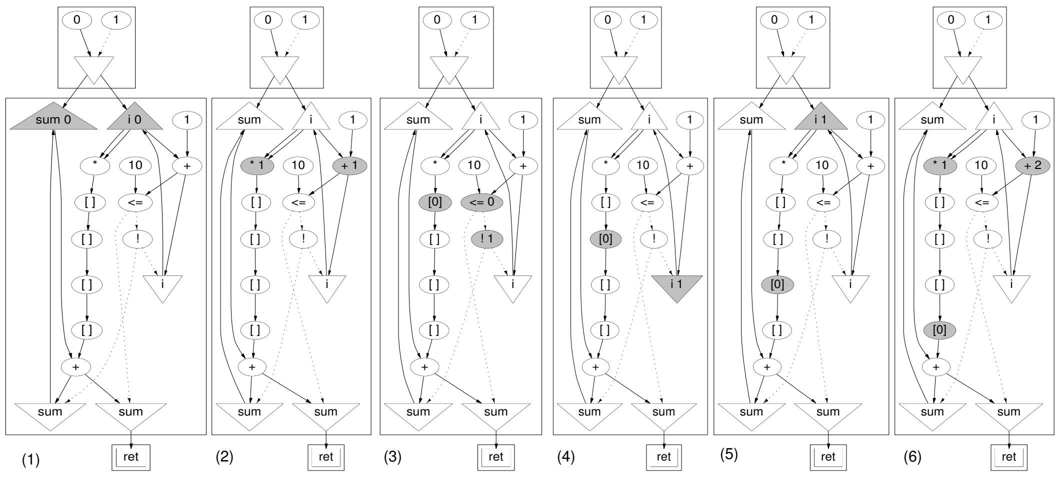

Figure 4: *Snapshots of the execution of the circuit in Figure 3. The shaded nodes are actively computing; they also indicate the current value of their output latch. We are assuming a 5-stage pipelined multiplier (each stage shown as [ ]); we assume all nodes in these graphs have the same latencies, except the Boolean negation, which takes zero time units (our implementation folds the inverter into the destination pipeline stage). In the last snapshot, two different values of* `i` *are simultaneously present in the multiplier pipeline.*

respond to *labels* in machine code, i.e., control-flow join points, which have a zero execution cost on a CPU.

This phenomenon is another facet of the tension between synchronization and parallelism. While a processor uses a program counter to sequence through the program, ASH relies on completely distributed control. MERGE and MUX operations are very simple forms of *synchronization*, used to merge several candidate values for a variable. Thus, the fine-grained parallelism of dataflow requires additional synchronization. This occurs even when the dataflow machine is not executed by an interpreter, but is directly mapped to hardware.

Distance to memory: a superscalar contains a limited number of load/store execution units (usually two). In flight memory access instructions have to be dynamically scheduled to access these units, but once they get hold of a unit they can initiate a memory access for a constant cost. (For example, the use of an on-processor LSQ allows write operations to complete in essentially zero time.)

In contrast, on ASH, each memory access operation is synthesized as a distinct hardware entity. Since our current implementation uses a monolithic memory, ASH requires the use of a network to connect the operations to memory. One such network implementation is described in Section 4.1.3. This network requires arbitration for the limited number of memory ports; the total arbitration cost is $O(\log(n))$ (n being the number of memory operations in the program). The wire length of such a network grows as $O(\sqrt{n})$.

The impact of the complexity of the memory network can be somewhat reduced by fragmenting memory in independent banks connected by separate networks, as we plan to do in future work.

Note that the asymptotic complexity of the memory itself has the same behavior: the decoders and selectors for a memory of n bits require $O(\log n)$ stages; the worst-case wire length is $O(\sqrt{n})$. This explains why memory systems grow intrinsically slower than processors in speed: today's memories are also bound by wire delays [4]. While ASH addresses some shortcomings of superscalar processors, it does not directly aim to solve the memory bottleneck problem; both models of computation attack this problem by trying to overlap memory stall time with useful computation.

Static vs. dynamic dataflow: in ASH, at most one instance of an operation may be executing at any given time, because each operation has a single output latch for storing the result. In contrast, a superscalar processor may have multiple instances of any instruction in flight at once, because the register renaming mechanism effectively provides a different storage element for each instance of an in-flight instruction. The only instruction which cannot be effectively pipelined without major changes in implementation is the LOAD: such operations have to wait for the memory access to complete before initiating a new access. A local reorder buffer could be employed for this purpose, but deviates from the spirit of ASH.

In ASH, loop unrolling and pipelining can sometimes provide similar results to the full dynamic dataflow model of superscalars, but are less general, since they are performed statically: we have seen instances where the CPU dynamic renaming outperformed the static version of the code for some input set.

Strict procedures: our current implementation of procedures relies on CALL nodes which are strict; i.e., to initiate a procedure all inputs to the node must be available. The fact that all inputs must be present before initiating a call introduces additional synchronization and puts the slowest argument computation on the dynamic critical path. When applicable, procedure inlining eliminates this problem as the procedure call network is specialized to become simple point-to-point channels.

In contrast, on a superscalar processor, procedure invocation is decoupled from passing of arguments (which are put into registers or on the stack) and the call is simply a branch. Thus, the code computing the procedure arguments does not need to complete before the procedure body is initiated. In fact, the computation of an unused procedure argument is never on the critical path.

The issues discussed above seem fundamental to ASH. Other shortcomings of ASH are attributable to policies in our compiler, and could be corrected by a more careful implementation.

4. FROM C TO LAYOUT

In this section we describe how Pegasus is translated to asynchronous circuits and we present detailed measurements of the synthesized circuits. We also discuss the reasons for the excellent power efficiency of our circuits.

4.1 CAB: The CASH Asynchronous Back-end

The asynchronous back-end of CASH translates the Pegasus representations into asynchronous circuits. The static dataflow ma-

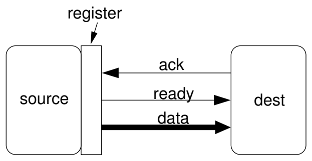

Figure 5: *Signaling protocol between data producers and consumers.*

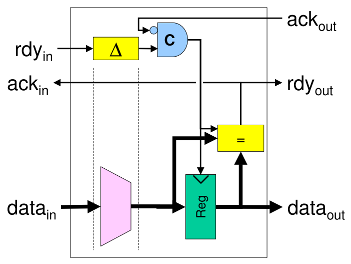

Figure 6: *Control circuitry for a pipeline stage.* Δ *is a* delay *matched to the computational unit. The block labeled with "="* *is a* completion detection *block, detecting when the register output has stabilized to a correct value.*

```
if (x > 0)
   y = -x;
else
   y = b*x;
```

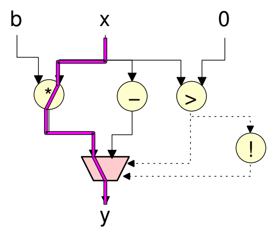

Figure 7: *Sample program fragment and the corresponding Pegasus circuit with the static critical path highlighted.*

chine semantics of Pegasus makes such a translation fairly straightforward. More details about this process are available in [107].

4.1.1 Synthesizing Scalar Computations

Pegasus representations could be mapped to asynchronous circuits in many ways. We have chosen to implement each Pegasus node as a separate hardware structure. Each IR node is implemented as a pipeline stage, using the micropipeline circuit style, introduced by Sutherland [98][3]. Each pipeline stage contains an output register which is used to hold the result of the stage computation. Each edge is synthesized as a *channel* consisting of three uni-directional signals as shown in Figure 5.

(1) A *data bus*, transfers the data from producer to consumer.

(2) A *data ready* wire from producer to consumer indicates when the data can be safely used by the consumer.

(3) An *acknowledgment* wire, from consumer to producer, indicates when the value has been used and the channel is $\perp$.

This signaling method, called the *bundled data protocol*, is widely employed in asynchronous circuits. The control circuitry driving a pipeline stage is shown in Figure 6. The "C" gate is a Müller C element [81], which implements the finite-state machine control for properly alternating data ready and acknowledgment signals. When there are multiple consumers the data bus is used to broadcast the value to all of them, and the channel contains one acknowledgment wire from each consumer. Due to the SSA form of Pegasus, each channel has a single writer. Therefore, there is no need for arbitration, making data transfer a lightweight operation.

Perhaps the most important feature of our implementation is the complete absence of any global control structures. Control is completely embodied in the handshaking signals—naturally distributed within the computation. This gives our circuits a very strong datapath orientation, making them amenable to efficient layout.

4.1.2 Lenient Evaluation

The form of speculative execution employed by Pegasus, which executes all forward branches simultaneously, alleviates the impact of branches, but may be plagued by the problem of *unbalanced paths* [8], as illustrated in Figure 7: the static critical path of the entire construct is the longest of the critical paths. If the short path is executed frequently, the benefits of speculation may be negated by the cost of the long path. This problem also occurs for machines which employ predicated execution. Traditionally this problem is addressed in two ways: (1) using profiling, only hyperblocks which ensure that the long path is most often executed at run-time are predicated, or (2) excluding certain hyperblock topologies from consideration, disallowing the predication of paths which differ widely in length.

Because there is no single PC, we can employ a third, and more elegant solution, in hardware by using *leniency* [91] to solve this problem. By definition, a lenient operation expects all of its inputs to arrive eventually, but it can compute its output using only a subset of its inputs. Lenient operators generate a result as soon as possible. For example, an AND operation can determine that the output is *false* as soon as one of its inputs is *false*.[4] While the output can be available before all inputs, our implementation ensures that a lenient operation sends an acknowledgment only after *all* of its inputs have been received. To obtain the full benefit of leniency one also needs to issue early acknowledgments, as suggested in [14]. In the asynchronous circuits literature, leniency was proposed under the name "early evaluation" [85]. Forms of lenient evaluation have been also been used in the design of arithmetic units for microprocessors: for example, some multiplier designs may generate the result very quickly when an input is zero.

MUXes are also implemented leniently: as soon as a selector is *true* and the corresponding data is available, a MUX generates its output. Note that it is crucial for the MUX to be decoded (see Section 2.2) in order for this scheme to work efficiently. A result of leniency is that *the dynamic critical path is the same as in a non-speculative implementation*. For example, if the multiplication in Figure 7 is not used, it does not affect the critical path.[5]

In addition to Booleans and multiplexors, all predicated operations are lenient in their predicate input. For example, if a LOAD operation receives a *false* predicate input, it can immediately emit an arbitrary output, since the actual output is irrelevant. It cannot, however, output a token until it receives its input token, since memory dependences are transitively implied. The irrelevant out-

[3] Unlike Sutherland's micropipelines, which used a 2-phase signalling protocol [12], we use 4-phase signaling, in which each signal returns to zero before initiating a new computation cycle.

[4] Lenient evaluation should not be confused with short-circuit evaluation: a short-circuit evaluation of an AND always evaluates the left operand, and if this one is *true*, it also evaluates the right one. However, a lenient evaluation generates a *false* result as soon as *either* input is known to be *false*.

[5] The multiplier can still be on the critical path because of its late acknowledgments, which may prevent the next wave of computation from propagating forward, as described in Section 3.3. This problem can be alleviated either by using a pipelined multiplier, or by using early acknowledgements [14].

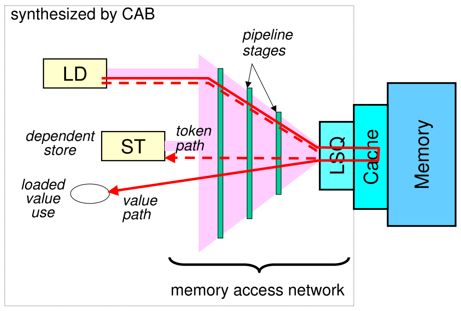

Figure 8: *Memory access network and implementation of the value and token forwarding network. The* LOAD *produces a data value consumed by the oval node. The* STORE *node may depend on the load (i.e., we have a token edge between the* LOAD *and the* STORE*, shown as a dashed line). The token travels to the root of the tree, which is a load-store queue (LSQ).*

put will be discarded downstream by a MUX or ETA node controlled by a *false* predicate.[6]

4.1.3 Memory Access

The most complicated part of the synthesis process is building the network used by the LOAD and STORE operations to access memory. Figure 8 illustrates how a load and a dependent store access memory through this network. Our current implementation consists of a hierarchy of buses and asynchronous arbiters used to mediate access to the buses. Memory instructions which are ready to access memory compete for these buses; the winners of the arbitration inject messages which travel up the hierarchy in a pipelined fashion. A memory operation can produce a token as soon as its effect is guaranteed to occur in the right order with respect to the potentially interfering operations. The network does not guarantee in-order message delivery, so by traveling to the root we maintain the invariant that a dependent operation will be issued only after all operations on which it depends have injected their requests in the LSQ. The root of the tree is a unique serialization point, guaranteeing in-order execution of dependent operations. The LSQ holds the description of the memory operations under execution until their memory effects are completed; it may also perform dynamic disambiguation and act as a small fully-associative cache. In Section 4.2.2 we discuss some disadvantages of this implementation. We currently synthesize a very simple load-store queue (LSQ), which can hold a single operation until its execution completes.

It is worthwhile to notice that this implementation of the memory access network is very much in the spirit of ASH, being completely distributed, composed entirely of pipeline stages, and using only control localized within each stage; it contains no global signals of any kind.

4.2 Low-level Evaluation

In this section we present measurements from a detailed low-level simulation. We synthesize C kernels into ASICs and evaluate their performance on standard data sets. Since CAB generates synthesizable Verilog, FPGAs could be targeted in principle for evaluation. There are two factors that prevent us from doing so: (1) commercial FPGAs are synchronous devices, and mapping some of the features of our asynchronous circuits would be very inefficient, (2) commercial FPGAs are not optimized for power [47]; they would thus probably negate one of the main advantages of our implementation scheme, the very low power consumption.

We use kernels from the Mediabench suite [66] to generate circuits. From each program we select one hot function (see Table 1) to implement in hardware (the only exception are the `g721` benchmarks, for which the hot function was very small, so we selected the function and one of its callers, we inlined the callee, unrolled the resulting loop and substituted references to an array of constants as inline constant values. The same code was used on the SimpleScalar simulator in comparisons.) The experimental results presented below are for the entire circuit synthesized by CAB, including the memory access network, but excluding the memory itself or I/O to the circuit. We report data only for the execution of each kernel, ignoring the rest of the program; due to long simulation times, we execute each kernel for the first three invocations in the program and we measure the cumulative values (time, energy, etc) for all three invocations. We do not estimate the overhead of invoking and returning from the kernel, since in this work we aim to understand the behavior of ASH, and not of a whole CPU+ASH system. Since our current back-end does not support the synthesis of floating-point computation we had to omit some kernels, such as the ones from the `epic`, `rasta` and `mesa` benchmarks.

The CAB back-end is used to generate a Verilog representation of each kernel. A detailed description of our methodology can be found in [107]. We use a 180nm/2V standard-cell library from STMicroelectronics, optimized for performance. The structural Verilog generated by our tool flow is partially technology-mapped by CAB and partially synthesized with Synopsys Design Compiler 2002.05-SP2. The technology-mapped circuits are placed-and-routed with Silicon Ensemble 5.3 from Cadence. Currently the placement is handled completely by Silicon Ensemble, operating on a flat netlist; we expect that CAB can use knowledge of the circuit structure to automatically generate floor-plans which can improve our results substantially[7]. Data collection with the commercial CAD tools for both `pegwit` benchmarks has failed after placement, so we present pre-placement numbers for these. (The performance for the other benchmarks is about 15% better than their pre-placement estimate.) Simulation is performed with Modeltech Modelsim SE5.7. We assume a perfect L1 cache, with a 600MHz cycle time. We synthesize a one-element LSQ for ASH.

Compilation time is on the order of tens of seconds for all these benchmarks, and is thus completely inconsequential compared to hardware synthesis through the commercial tool-chain (the worst-case program takes about 30s through CASH, one hour through synthesis and more than five hours for place-and-route). The code expansion in terms of lines of code from C to Verilog is a factor of 200x. All the results in this section are obtained without loop unrolling, which can increase circuit area and compilation time.

4.2.1 Area

Figure 9 shows the area required for each of these kernels. The area is broken down into "computation" and "memory tree." The memory tree is the area of the arbiter circuits used to mediate access to memory and of the hierarchical pipelined buses. For reference, in the same technology a minimal RISC core can be synthesized in 1.3mm^2, a 16×16 multiplier requires 0.1mm^2, and a complete P4 processor die, including all caches, has 217mm^2. This shows that while the area of our kernels is sometimes substantial, it is certainly affordable, especially in future technologies. Normalizing the area

[6]The predicated-*false* operation does not need to swing the output lines, it need only assert the *data ready* signal (see Section 4.1.1). This will decrease the power consumption.

[7]Good placement and physical optimizations can account for as much as a factor of 14x in size and 2.3x in performance [36]!

Benchmark	Function	Lines
adpcm_d	adpcm_decoder	80
adpcm_e	adpcm_coder	103
g721_d	fmult+quan	41
g721_e	fmult+quan	41
gsm_d	Short_term_synthesis_filtering	24
gsm_e	Short_term_analysis_filtering	45
jpeg_d	jpeg_idct_islow	241
jpeg_e	jpeg_fdct_islow	144
mpeg2_d	idctcol	55
mpeg2_e	dist1	92
pegwit_d	squareDecrypt	78
pegwit_e	squareEncrypt	77

Table 1: *Embedded benchmark kernels used for the low-level measurements and their size in original (un-processed) source lines of code. For* `g721` *the function* `quan` *was inlined into* `fmult`.

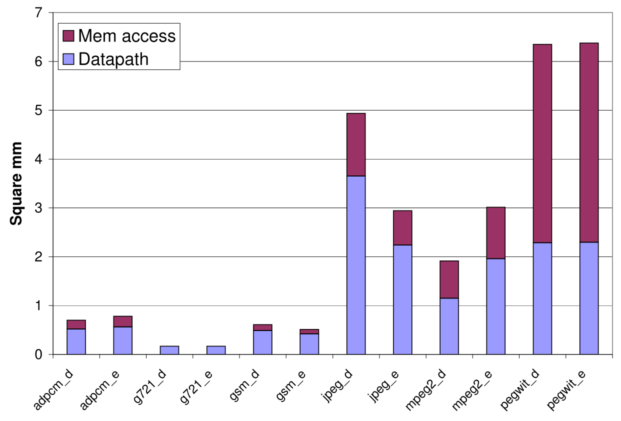

Figure 9: *Silicon real-estate in mm^2 for each kernel.*

versus the object file size, we require on average 0.87mm^2/kb of a gcc-generated MIPS object file.

4.2.2 Execution performance

Figure 10 shows the normalized execution time of each kernel against a baseline 600MHz4-wide superscalar processor. While we did not simulate a VLIW, we expect the trends to be similar, since the superscalar maintains a high IPC for these kernels. The processor has the same perfect L1 cache, but a 32-element LSQ. On average, ASH circuits are 1.33 times slower, but 4 kernels are faster than on the processor.

Given the unlimited amount of ILP that can be exploited by ASH, these results are somewhat disappointing. An analysis of ASH circuits has pointed out that, although these circuits can be improved in many respects, the main bottleneck of our current design is the memory access protocol. In our current implementation, as described in Section 4.1.3, a memory operation does not release a token until its request has reached memory (i.e., the token must traverse the network end-to-end in both directions). An improved construction would allow an operation to (1) inject requests in the network, allowing them to travel out-of-order, and (2) release the token to the dependent operations immediately. The network packet can carry enough information to enable the LSQ to buffer out-of-order requests and to execute the memory operations in the original program order. This kind of protocol is actually used by superscalar processors, which inject requests in order in the load-store queue, and can proceed to issue more memory operations before the previous ones have completed.

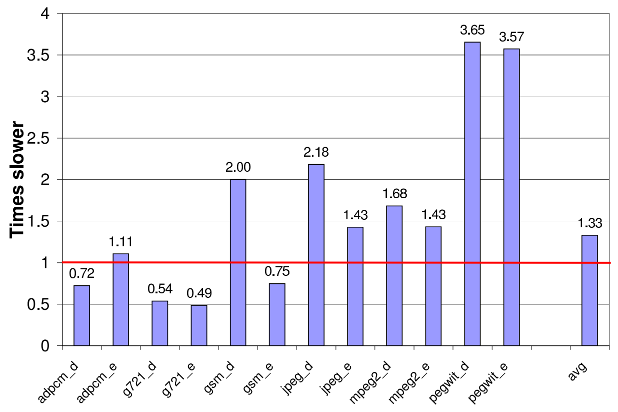

Figure 10: *Kernel slowdown compared to a 4-wide issue 600MHz superscalar processor in 180nm. A value of 1 indicates identical performance, values bigger than 1 indicate slower circuits on ASH.*

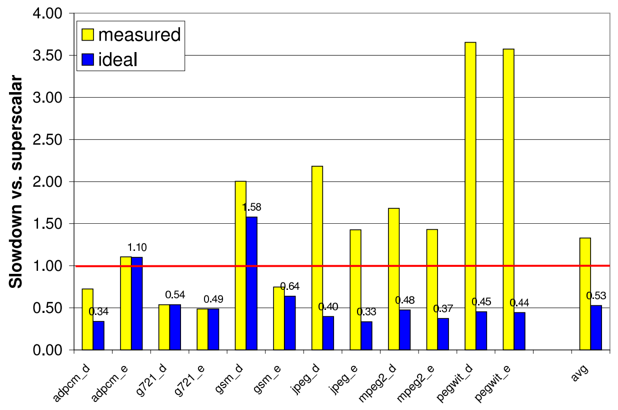

Figure 11: *Evaluation the impact of an ideal memory interconnection protocol. The left bar reproduces the data from Figure 10.*

To gauge the impact of the memory network on program performance, we performed a limit study using a behavioral implementation of the network in which each stage has zero latency. The improvement in performance is shown in Figure 11: programs having large memory access networks in Figure 9 display significant improvements (up to 8x for `pegwit`) which shows that programs which perform many memory accesses are bound by the memory network round-trip time. These numbers are obtained assuming that both value and token travel very quickly through the network; in reality, we can only substantially speed-up the token path, so the performance of a better protocol still has to be evaluated.

In Figure 12 we measure ASH performance using several MIPS metrics: the bottom bar we labeled MOPS, for millions of *useful* arithmetic operations per second. The incorrectly speculated arithmetic is accounted for as MOPSspec. Finally, MOPSall includes "auxiliary" operations, including the MERGE, ETA, MUX, COMBINE, pipeline balancing FIFOs, and other overhead operations. Although speculative execution sometimes dominates useful work (e.g., `g721`), on average 1/3 of the executed arithmetic operations are incorrectly speculated.[8] For some programs the control operations constitute a substantial fraction of the total number of executed operations. On average our programs sustain 1 GOPS.

[8]The compiler can control the degree of speculation by varying the hyperblock construction algorithm; we have not yet explored the trade-offs involved.

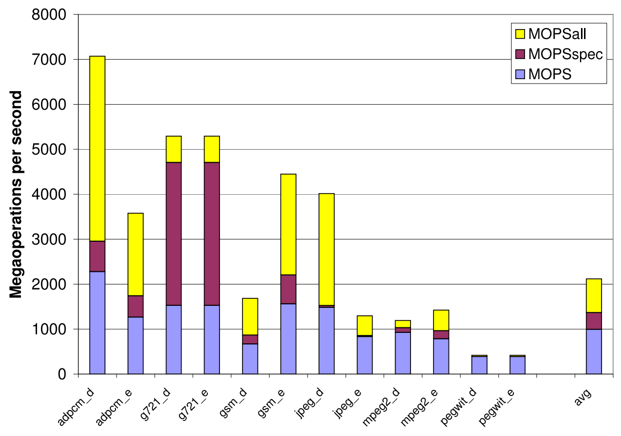

Figure 12: *Computational performance of the ASH Mediabench kernels, expressed in millions of operations per second (MOPS).*

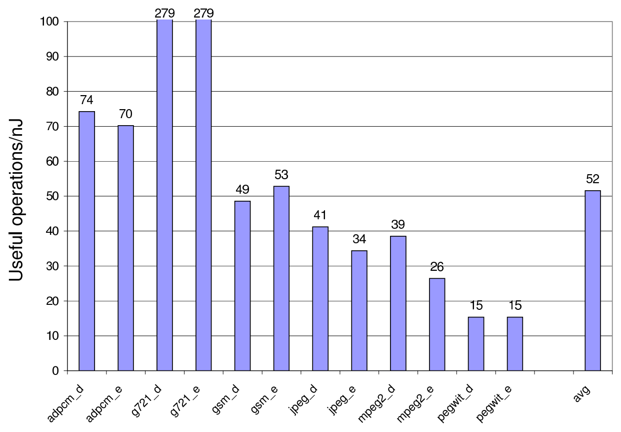

Figure 13: *Energy efficiency of the synthesized kernels, expressed in useful arithmetic operations per nanoJoule.*

4.2.3 Power and Energy

The power consumption of our circuits ranges between 5.5mW (`g721`) and 35.9mW (`jpeg_d`), with an average of 19.3mW. In contrast, a very low power DSP chip in the same technology draws about 110mW.

In order to quantify energy efficiency we use the normalized metric proposed by Claasen [29], MOPS/mW, or, equivalently, operations/nanoJoule. Figure 13 shows how our kernels fare from this perspective. We count only the useful operations in the MIPS metric, i.e., the bottom bar from Figure 12. The energy efficiency for CASH systems is between 15 and 280 operations/nJ, with an average of 52. `g721_d` and `g721_e` are outliers because they do not make any memory accesses, and their implementation is thus extremely efficient. For comparison Figure 14 shows, on a logarithmic scale, the energy efficiency of microprocessors, digital signal processing, custom hardware circuits from [116], an asynchronous microprocessor [77], and FPGAs. All these circuits use comparable hardware technologies (180 and 250nm). ASH is three to four orders of magnitude better than a superscalar microprocessor and between one and two orders of magnitude better than a DSP.

Figure 15 compares ASH with a 4-wide low-power superscalar (modeled with Wattch [15], using aggressive clock-gating for power reduction, and drawing around 5W dynamic power) in terms of energy-delay [53], a metric which is relatively independent of execution speed. ASH circuits are between 16 and 3600 times better.

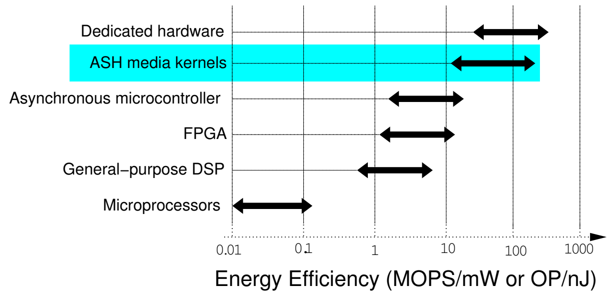

Figure 14: *Energy efficiency of several computational models using comparable technologies.*

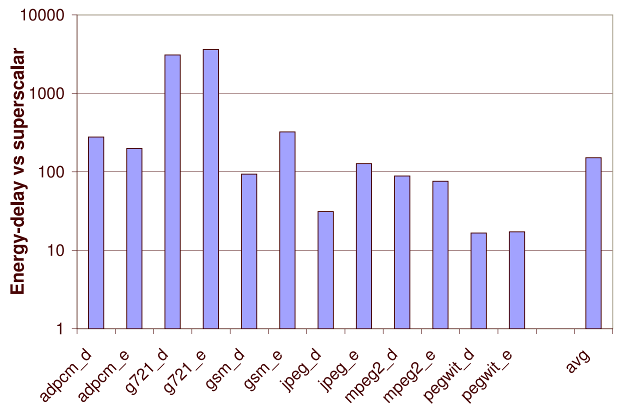

Figure 15: *Energy-delay ratio between ASH and superscalar.*

The circuits without memory access are again substantially better (one order of magnitude) than the other.

Power is roughly proportional to the circuit activity; the power wasted on speculation should be proportional to the number of speculative operations executed. Currently we believe that we should negotiate the power-performance trade-off in the direction of expending more power to increase performance: since power is very low, some increase in power is acceptable even for a relatively small increase in performance.

4.2.4 Discussion

There are multiple sources of inefficiency in the implementation of superscalar processors, which account for the huge difference in power and energy.

(1) The clock distribution network for a large chip accounts for up to 50% of the total power budget; while some of this power is spent on the latches, which exist also in asynchronous implementations [12], the big clock network and its huge drivers still account for a substantial fraction of the power.

(2) Most of the pipeline stages in ASH are usually inactive (see for example Figure 4). Since these circuits are asynchronous, they only draw static power when inactive, which is very small in a .18μm technology. While leakage power is a big issue in future technologies, there are many circuit- and architecture-level techniques [103] that can be applied to reduce its impact in ASH.

(3) On the Pentium 4 die, even excluding caches, all functional units combined (integer, floating-point and MMX) together take less than 10% of the chip area. The remaining 90% is devoted entirely to mechanisms that only support the computation, without producing "useful" results. From the point of view of the energy efficiency metric we use, all of this extra activity is pure overhead.

(4) [77] suggests that more than 70% of the power of the asynchronous Lutonium processor is spent in instruction fetch and decode. This shows that there is an inherent overhead for *interpreting* programs encoded in machine-code, which ASH does not have to pay.

(5) Finally, in a microprocessor, and even in many ASICs, functional units are heavily optimized for speed at the expense of power and area. Since SC replicates lavishly functional units, the trade-off has to be biased in the reverse direction.

It has been known that dedicated hardware chips can be vastly more energy-efficient that processors [116]. But most often the algorithms implemented in dedicated hardware have a natural high degree of parallelism. This paper shows for the first time that a fully automatic tool-chain starting from C programs can generate results comparable in speed with high-end processors and in power with custom hardware.

We are confident that further optimizations, including circuit-level techniques, high-level compiler transformations, a better memory access protocol, and compiler-guided floor-planning, will substantially increase ASH's performance.

5. RELATED WORK

Optimizing compilers. Pegasus is a form of dataflow intermediate language, an idea pioneered by Dennis [39]. The crux of Pegasus is handling memory dependences without excessively inhibiting parallelism. The core ideas on how to use tokens to express fine-grained location-based synchronization were introduced by Beck, et al. [11]. The explicit representation of memory dependences between program operations has been suggested numerous times in the literature, e.g., Pingali's Dependence Flow Graph [83]; or Steensgaard's adaptation of Value-Dependence Graphs [97]. Other researchers have also explored extending SSA to handle memory dependences, e.g.,[35, 44, 64, 28, 30, 71, 72]. But none of the approaches is as simple as ours.

The integration of predication and SSA has also been done in PSSA [24, 25]. PSSA, however, does not use ϕ functions and therefore loses some of the appealing properties of SSA. Our use of the hyperblock as a basic optimization unit and our algorithm for computing block and path predicates were inspired by this work.

High-level synthesis. While there is a substantial amount of research on hardware synthesis from high-level languages, and dialects of C and C++, none of it supports C as fully as CASH does. One major difference between our approach and the vast majority of other efforts is that we use dusty-deck C while most other projects employ C as a hardware description language (HDL). They either add constructs (e.g., reactivity, concurrency, and variable bit-width) to C in order to make it more suitable to expressing hardware properties, or/and remove constructs (e.g., pointers, dynamic allocation and recursion) that do not correspond to natural hardware objects, obtaining "Register Transfer C." Other efforts impose a strict coding discipline in C in order to obtain a synthesizable program. There are numerous research and commercial products using variants of C as an HDL [80]: Pico from HP [93], Cynlib, originally from Cynapps, now at Forte Design Systems [88], Cyber from NEC [109], A|RT builder originally from Frontier Design, now at Xilinx [59], Scenic/CoCentric from Synopsys [69], N2C from CoWare [32], compilers for Bach-C (originally based on Occam) from Sharp [60], OCAPI from IMEC [90], Synopsys' c2verilog, originally from C-Level Design [96], Celoxica's Handel-C compiler [31], and Cadence's ECL (based on C and Esterel) compiler [65], SpC from Stanford [94] and also [48, 6, 54, 55].

Our goals are most closely related to the "chip-in-a-day" project from Berkeley [37], but our approach is very different: that project starts from parallel Statechart descriptions and employs sophisticated hard macros to synthesize synchronous designs with automatic clock gating. In contrast we start from C, use a small library of standard cells, and build asynchronous circuits.

Reconfigurable computing. A completely different approach to hardware design is taken in the research on reconfigurable computing, which relies on automatic compilation of C or other high-level languages to target reconfigurable hardware [112]. Notable projects are: PRISM II [110], PRISC [84], DISC [113], NAPA [49], DEFACTO [40], Chimaera [115], OneChip [114], RaPiD [41], PamDC [102], StreamC [50], WASMII [100], XCC/PACT [23], PipeRench [51], RAW/Virtual Wires [9], the systems described in [95] and [78], compilation of Term-Rewriting Systems [58], or synthesis of dataflow graphs [86].

CASH has been influenced by the work of Callahan on the GARP C compiler [22, 68]. GarpCC is still one of the few systems with the broad scope of targeting C to a reconfigurable fabric.

None of these approaches targets a true Spatial Computation model, with completely distributed computation and control.

Dataflow machines. A large number of dataflow machine architectures have been proposed and built; see for example a survey in [106]. All of these were interpreters, executing programs described in a specialized form of machine code. Most of the dataflow machines were programmed in functional languages, such as VAL and Id. CASH could in principle be used to target a traditional dataflow machine. To our knowledge, none of the other efforts to translate C for execution on dataflow machines (e.g., [105]) reached completion.

Asynchronous circuit design. Traditionally, asynchronous synthesis has concentrated on synthesizing individual controllers. However, in the last decade, a number of other approaches have also explored the synthesis of entire systems. The starting point for these related synthesis flows is a high-level language (usually an inherently parallel one based on Hoare's Communicating Sequential Processes [57]) suitable for describing the parallelism of the application, e.g., Tangram [104], Balsa [42], OCCAM [79], CHP [76, 101]. With the exception of [76], synthesis tends to follow a two step approach: the high-level description is translated in a syntax-directed fashion into an intermediate representation; then, each module in the IR is (template-based) technology mapped into gates. An optional optimization step [27] may be introduced to improve the performance of the synthesized circuits. In [76], the high-level specification is decomposed into smaller specifications, some dataflow and local optimizations techniques are applied, and the resulting low-level specifications are synthesized in a template-based fashion.

Our approach is somewhat similar to the syntax-directed flows with three notable differences. First, our input language is a well-established, imperative, sequential programming language; handling pointers and arrays is central in our approach. Second, the CASH compiler performs extensive analysis and optimization steps before generating the intermediate form. Finally, the target implementation for our intermediate form consists of *pipelined* circuits, which naturally increase performance when compared with traditional syntax-directed approaches.

Spatial Computation. Various models related to Spatial Computation have been investigated by other research efforts: compiling for finite-sized fabrics has been studied by research on systolic arrays [62], RAW [67], PipeRench [52] and TRIPS [89]. More remotely related are efforts such as SmartMemories [75] and Imagine [87]. Unlimited or virtualized hardware is exploited in proposals such as SCORE [26], [38], and WaveScalar [99]. Among the latter efforts, our research is distinguished by the fact that (1) it targets purely spatial implementations, without centralized resources and control and (2) has produced the most detailed end-to-end evaluation of such an architecture.

6. CONCLUSIONS

We have investigated one particular instance of the ISA-less class of Spatial Computation models, ASH, Application-Specific Hardware. In ASH a software program is translated directly into an executable hardware form, which has no interpretation layer. The synthesized hardware closely reflects the program structure—the definer-user relationships between program operations are translated directly into point-to-point communication links connecting arithmetic units. The resulting circuit is completely distributed, featuring no global communication or broadcast, no global register files, no associative structures, and using resource arbitration only for accessing global memory. While the use of a monolithic memory does not take advantage of the freedom provided by such an architecture, it substantially simplifies the completely automatic implementation of C language programs. We have chosen to synthesize dynamically self-scheduled circuits, i.e., where computations are carried out based on availability of data and not according to a fixed schedule. A natural vehicle for implementing self-scheduled computations was provided by asynchronous hardware.

A detailed investigation of the run-time behavior of the spatial structures has shown that their distributed nature forces them to incur non-negligible overheads when handling control-intensive programs. In contrast, traditional superscalar processors, with their global structures (such as branch prediction, control speculation and register renaming), can more efficiently execute control-intensive code. Since ASH complements the capabilities of traditional monolithic processors, a promising avenue of research is the investigation of hybrid computation models, coupling a monolithic and distributed engine.

Circuit-level simulations of ASH have shown that it provides very good performance when used to implement programs with high ILP, such as media kernels. The energy efficiency of ASH is more than one to two orders of magnitude better when compared even to low-power DSP processors and three or more orders of magnitude better than general purpose superscalar microprocessors. Our compilation methodology for ASH indicates that it is possible to convert even dusty-deck programs written in a high-level, imperative, sequential language into efficient hardware. We believe this work is the beginning of a line of research that will eliminate the designer productivity gap, decrease the power problem, and allow for the exploitation of the promised massive numbers of devices that will become available in future technologies.

7. ACKNOWLEDGEMENTS

This research is funded in part by the National Science Foundation under Grants No. CCR-9876248 and CCR-0205523 by DARPA under contracts MDA972-01-03-0005 and N000140110659 and by the SRC. The procedure inlining Suif pass was written by Tim Callahan. The loop unrolling pass was written by Todd Mowry. Some optimizations were implemented by Pedro Artigas. We thank Dan Vogel for help with scripting and benchmark management. Finally, we wish to thank the many reviewers for their helpful comments.

8. REFERENCES

[1] International technology roadmap for semiconductors (ITRS). http://public.itrs.net/Files/1999_SIA_Roadmap/Design.pdf, 1999.

[2] 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 *International Symposium on Computer Architecture (ISCA)*, June 2000.

[3] Vicki H. Allan, Reese B. Jones, Randal M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[4] Bharadwaj S Amrutur and Mark A Horowitz. Speed and power scaling of SRAMs. *IEEE Journal of Solid State Circuits*, 35(2):175–185, February 2000.

[5] Andrew W. Appel. SSA is functional programming. ACM SIGPLAN Notices, April 1998.

[6] Guido Arnout. C for system level design. In *Design, Automation and Test in Europe (DATE)*, pages 384–387, Munich, Germany, March 1999.

[7] Arvind and Robert A. Iannucci. A critique of multiprocessing von Neumann style. In *International Symposium on Computer Architecture (ISCA)*, pages 426–436. IEEE Computer Society Press, 1983.

[8] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *International Symposium on Computer Architecture (ISCA)*, December 1997.

[9] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1999.

[10] Daniel W. Bailey and Bradley J. Benschneider. Clocking design and analysis for a 600-MHz Alpha microprocessor. *IEEE Journal of Solid-State Circuits*, 33(11):1627, November 1998.

[11] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[12] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer Verlag, 1995. summary at www.cse.ttu.edu.tw/ cheng/courses/soc/S02/AsyncSoc08.ppt; also Nat.Lab. Technical Note Nr. UR 005/94, Philips Research Laboratories, Eindhoven, the Netherlands.

[13] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and C. McMullin. *Logic Minimization Algorithms for Digital Circuits*. Kluwer Academic Publishers, Boston, MA, 1984.

[14] C.F. Brej and J.D. Garside. Early output logic using anti-tokens. In *International Workshop on Logic Synthesis*, pages 302–309, May 2003.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture (ISCA)*, pages 83–94. ACM Press, 2000.

[16] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[17] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2002.

[18] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, San Francisco, CA, March 23-26 2003.

[19] Mihai Budiu and Seth Copen Goldstein. Inter-iteration scalar replacement in the presence of conditional control-flow. Technical Report CMU-CS-04-103, Carnegie Mellon University, Department of Computer Science, 2004.

[20] Mihai Budiu, Mahim Mishra, Ashwin Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–66, Napa Valley, CA, April 2002.

[21] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25, pages 13–25. ACM SIGARCH, June 1997.

[22] Timothy J. Callahan and John Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *International Conference on Field Programmable Logic and Applications (FPL)*, volume 1482 of *Lecture Notes in Computer Science*, Tallinin, Estonia, September 1998. Springer-Verlag.

[23] João M. P. Cardoso and Markus Weinhardt. PXPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In *International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier (La Grande-Motte), France, September 2002.

[24] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[25] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[26] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy, André DeHon, and John Wawrzynek. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *International*

Conference on Field Programmable Logic and Applications (FPL), Lecture Notes in Computer Science. Springer Verlag, 2000.

[27] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *DAC*, pages 405–410, New York, June 10–14 2002. ACM Press.

[28] Fred Chow, Raymond Lo, Shin-Ming Liu, Sun Chan, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction (CC)*, pages 253–257, April 1996.

[29] T.A.C.M. Claasen. High speed: not the only way to exploit the intrinsic computational power of silicon. In *IEEE International Solid-State Circuits Conference*, pages 22–25, San Francisco, CA, 1999. IEEE Catalog Number: 99CH36278.

[30] Keith D. Cooper and Li Xu. An efficient static analysis algorithm to detect redundant memory operations. In *Workshop on Memory Systems Performance (MSP '02)*, Berlin, Germany, June 2002.

[31] Celoxica Corporation. Handel-C language reference manual, 2003.

[32] CoWare, Inc. Flexible platform-based design with the CoWare N2C design system, October 2000.

[33] David E. Culler and Arvind. Resource requirements of dataflow programs. In *International Symposium on Computer Architecture (ISCA)*, pages 141–150, 1988.

[34] 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 (TOPLAS)*, 13(4):451–490, 1991.

[35] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 36–45. ACM Press, 1993.

[36] W. J. Dally and A. Chang. The role of custom design in ASIC chips. In *Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[37] W. R. Davis, N. Zhang, K. Camera, D. Markovic, T. Smilkstein, M. J. Ammer, E. Yeo, S. Augsburger, B. Nikolic, and R. W. Brodersen. A design environment for high throughput, low power dedicated signal processing systems. *IEEE Journal of Solid-State Circuits*, 37(3):420–431, March 2002.

[38] André DeHon. Very large scale spatial computing. In *Third International Conference on Unconventional Models of Computation*, 2002.

[39] Jack B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science 19: Programming Symposium*, pages 362–376. Springer-Verlag: Berlin, New York, 1974.

[40] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Bridging the gap between compilation and synthesis in the DEFACTO system. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[41] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997.

[42] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer J.*, 45(1):12–18, 2002.

[43] Brian Fields, Rastislav Bodík, and Mark D.Hill. Slack: Maximizing performance under technological constraints. In *International Symposium on Computer Architecture (ISCA)*, pages 47–58, 2002.

[44] David Mark Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1995.

[45] Emden Gansner and Stephen North. An open graph visualization system and its applications to software engineering. *Software Practice And Experience*, 1(5), 1999. http://www.research.att.com/sw/tools/graphviz.

[46] Guang R. Gao. *A Pipelined Code Mapping Scheme for Static Data Flow Computers*. PhD thesis, MIT Laboratory for Computer Science, 1986.

[47] Varghese George, Hui Zhang, and Jan Rabaey. The design of a low energy FPGA. In *International Symposium on Low-Power Design (ISLPED)*, pages 188–193. ACM Press, 1999.

[48] A. Ghosh, J. Kunkel, and S. Liao. Hardware synthesis from C/C++. In *Design, Automation and Test in Europe (DATE)*, pages 384–387, Munich, Germany, March 1999.

[49] M. Gokhale and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In W. Moore and W. Luk, editors, *International Conference on Field Programmable Logic and Applications (FPL)*, pages 399–408, Oxford, England, August 1995. Springer.

[50] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, 2000.

[51] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial computing using molecular electronics. In *International Symposium on Computer Architecture (ISCA)*, pages 178–189, Göteborg, Sweden, 2001.

[52] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *International Symposium on Computer Architecture (ISCA)*, pages 28–39, Atlanta, GA, 1999.

[53] R. Gonzalez and M. Horowitz. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits*, 32(8), August 1997.

[54] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, Alex Nicolau, Timothy Kam, Michael Kishinevsky, and Shai Rotem. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference (DAC)*, pages 898–903. ACM Press, 2002.

[55] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil D. Dutt, Rajesh K. Gupta, and Alexandru Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference (DAC)*, pages 269–272, 2001.

[56] R. Ho, K. Mai, and M. Horowitz. The future of wires. *IEEE Journal*, 89(4):490–504, April 2001.

[57] Hoare. Communicating sequential processes. In *C. A. A. Hoare and C. B. Jones (Ed.), Essays in Computing Science, Prentice Hall*. 1989.

[58] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, San Jose, California, November 2000.

[59] Doug Johnson. Programming a Xilinx FPGA in "C". *Xcell Quarterly Journal*, 34, 1999.

[60] Andrew Kay, Toshio Nomura, Akihisa Yamada, Koichi Nishida, Ryoji Sakurai, and Takashi Kambe. Hardware synthesis with Bach system. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Orlando, 1999.

[61] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, 2 edition, 1988.

[62] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[63] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer Architecture (ISCA)*, 1992.

[64] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *the 1998 International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 128–143, March 1998.

[65] Luciano Lavagno and Ellen Sentovich. ECL: A specification environment for system-level design. In *Design Automation Conference (DAC)*, pages 511–516, New Orleans, LA, June 1999.

[66] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[67] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, 1998.

[68] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Design Automation Conference (DAC)*, 2000.

[69] Stan Liao, Steven W. K. Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Design Automation Conference (DAC)*, pages 70–75, 1997.

[70] Andrew Matthew Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, Computer Science Department, 1995. CS-TR-95-21.

[71] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37. ACM Press, 1998.

[72] John Lu and Keith D. Cooper. Register promotion in C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM Press, 1997.

[73] Scott A. Mahlke, Richard E. Hauk, James E. McCormick, David I. August, and Wen mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149, Santa Margherita Ligure, Italy, May 1995. ACM.

[74] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[75] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *International Symposium on Computer Architecture (ISCA)*, June 2000.

[76] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[77] Alain J. Martin, Mika Nystrm, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, and Ahmet Tura. The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2003.

[78] Tsutomu Maruyama and Tsutomu Hoshino. A C to HDL compiler for pipeline processing on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2000.

[79] D. May. OCCAM. *SIGPLAN Notices*, 18(4):69–79, May 1983.

[80] Giovanni De Micheli. Hardware synthesis from C/C++ models. In *Design, Automation and Test in Europe (DATE)*, Munich, Germany, 1999.

[81] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *International Symposium on the Theory of Switching Functions*, pages 204–243, 1959.

[82] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, 1990.

[83] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *ACM Symposium on Principles of Programming Languages (POPL)*, volume 18, 1991.

[84] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmed functional units. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 172–180, November 1994.

[85] Robert B. Reese, Mitch A. Thornton, and Cherrice Traver. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *International Conference on Computer Design (ICCD)*, page 18, Austin, TX, September 23-26 2001.

[86] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and A.P.W. Böhm. An automated process for compiling dataflow graphs into hardware. *IEEE Transactions on VLSI*, 9 (1), February 2001.

[87] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 1998.

[88] Ray Roth and Dinesh Ramanathan. A high-level design methodology using C++. In *IEEE International High Level Design Validation and Test Workshop*, November 1999.

[89] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A technology-scalable architecture for fast clocks and high ILP. In *Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[90] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A programming environment for the design of complex high speed ASICs. In *Design Automation Conference (DAC)*, pages 315–320, San Francisco, June 1998.

[91] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs require? In *International Conference on Functional Programming Languages and Computer Architecture*, pages 216–225. ACM Press, 1995.

[92] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C.L. Thompson. Compilers for instruction-level parallelism. *IEEE Computer*, 30(12):63–69, 1997. This was a report from a cross-industry task force on ILP.

[93] R. Schreiber, S. Aditya (Gupta), B.R. Rau, S. Mahlke, V. Kathail, B. Ra. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 2001.

[94] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on VLSI*, 2001.

[95] Greg Snider, Barry Shackleford, and Richard J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 115–124. ACM Press, 2001.

[96] Donald Soderman and Yuri Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 339–342, Los Alamitos, CA, April 1998. IEEE Computer Society Press.

[97] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

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

[99] Steven Swanson, Ken Michelson, and Mark Oskin. WaveScalar. Technical Report 2003-01-01, Washington University at Seattle, Computer Science Department, January 2003.

[100] A. Takayama, Y. Shibata, K. Iwai, H. Miyazaki, K. Higure, and X.-P. Ling. Implementation and evaluation of the compiler for WASMII, a virtual hardware system. In *International Workshop on Parallel Processing*, pages 346–351, 1999.

[101] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate oncurrent pipeline synthesis. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 17–27, Heraklion, Crete, Greece, April 2004.

[102] Hervé Touati and Mark Shand. PamDC: a C++ library for the simulation and generation of Xilinx FPGA designs. http://research.compaq.com/SRC/pamette/PamDC.pdf, 1999.

[103] Y-F. Tsai, D. Duarte, N. Vijaykrishnan, and M.J. Irwin. Implications of technology scaling on leakage reduction techniques. In *Design Automation Conference (DAC)*, San Diego, CA, June 2004.

[104] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, volume 5 of *Intl. Series on Parallel Computation*. Cambridge University Press, 1993.

[105] A. H. Veen and R. van den Born. The RC compiler for the DTN dataflow computer. *Journal of Parallel and Distributed Computing*, 10:319–332, 1990.

[106] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.

[107] Girish Venkataramani, Mihai Budiu, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic Syntheiss*, Temecula, CA, June 2004.

[108] John von Neumann. First draft of a report on the EDVAC. Contract No. W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. Reprinted (in part) in Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, June 1945.

[109] Kazutoshi Wakabayashi and Takumi Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design*, 19(12):1507–1522, December 2000.

[110] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, Napa Valley, CA, Apr 1993.

[111] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[112] Niklaus Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31 (6):25–31, June 1998.

[113] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 99–107, Napa, CA, April 1995.

[114] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In J. Arnold and K. L. Pocek, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 126–135, Napa, CA, April 1996.

[115] Alex Zhi Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. In *International Symposium on Computer Architecture (ISCA)*, ACM Computer Architecture News. ACM Press, 2000.

[116] Ning Zhang and Bob Brodersen. The cost of flexibility in systems on a chip design for signal processing applications . http://bwrc.eecs.berkeley.edu/Classes/EE225C/Papers/arch_design.doc, Spring 2002.