Spatial Computation
— Summary of the Ph.D. Thesis —

Mihai Budiu

December 2003

1 Thesis Statement

Software compilation technology for targeting predicated architectures can be naturally adapted for performing the automatic synthesis of application-specific, custom hardware dataflow machines. This compilation methodology translates media processing kernels into hardware with a high degree of instruction-level and pipeline parallelism. However, the resulting distributed computation structures are not as easily amenable as traditional monolithic superscalar processors at using acceleration mechanisms such as (correlated) prediction, speculation and register renaming. The lack of these mechanisms is a severe handicap when executing control-intensive code.

2 Motivation

For most computer users the speed and capabilities of today's computer systems were undreamed of as little as a decade ago. The relentless advance of technology at an exponential pace for more than 40 years has produced amazingly complex and powerful machines: in September 2003 Intel released the Itanium 2 processor for servers, built out of 410 million transistors on a single 374mm^2 chip and consuming 130W. A desktop microprocessor with more than 1 billion transistors is expected before 2007. The exponential increase in computing resources is expected to last at least another decade, and perhaps even more if the promises of nanotechnology come true.

However, all this progress has not been for free. The very success of miniaturization creates new unexpected obstacles. Figure 1 is a slide from a presentation given by Michael Flynn at the Federated Computer Research Conference 2003 [7], illustrating one such difficulty: the complexity of the hardware designs increases with the number transistors, at an exponential pace. For instance, the complexity of a hardware design depends on the number of *exceptions* that cannot be automatically handled by Computer-Aided Design (CAD) tools; the number of these exceptions grows with the number of transistors. Although the productivity of hardware designers increases too, due to improvements in CAD tools, it does so at a slower pace. The end result is a growing *productivity gap* between what we have and what we can economically use. Fewer and fewer companies can close this gap, and when they do, they employ larger and larger design, test, verification and manufacturing teams.

The problems of hardware design productivity are nowhere near of being solved; in fact more ominous obstacles are surfacing. Figure 2 is another slide taken from the same presentation, comparing the signal propagation delay through logic gates and "average-sized" wires. Since many structures of a modern processor, such as the forwarding paths on the pipeline, the multi-ported register files, the instruction wake-up

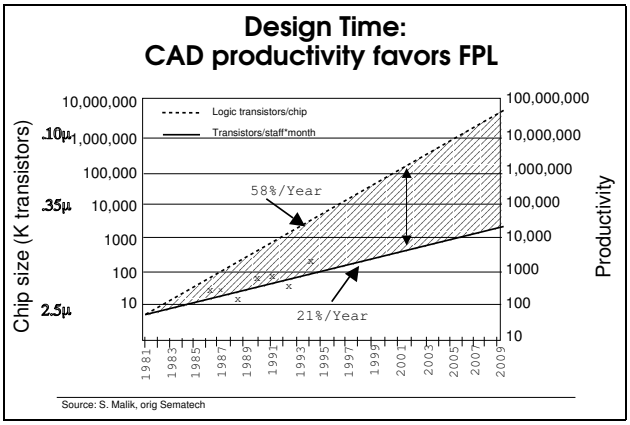

Figure 1: *Complexity and productivity versus technology generation.*

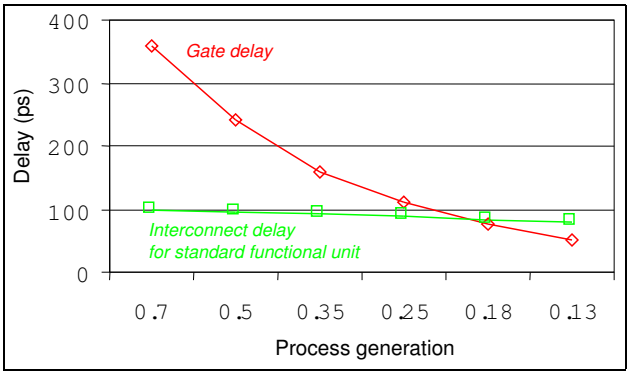

Figure 2: *Wires and not gates dominate the delay in advanced technologies.*

logic, etc., require global signals (i.e., spanning multiple modules), there simply is not enough time for the signals to propagate along these wires at very high clock speeds. In fact, at 10Ghz a signal can cover less than 1% of the surface of a chip in one clock cycle [1], making any architecture relying on global signals infeasible.

The research presented in this document is aimed directly at these problems. We explore **(A)** a new computational model, which requires no global synchronization, and **(B)** a new CAD methodology, aimed at bridging both software compilation and microarchitecture. Our model is named Spatial Computation. In Spatial Computation applications written in high-level languages are compiled directly into hardware circuits. The synthesized circuits feature only localized communication, require no broadcast, no global control, and are timing-insensitive, and thus correct even when the communication latency is not statically predictable. The compiler we have developed requires no designer intervention (i.e., it is fully automatic), is fast, and exploits both instruction-level parallelism and pipelining. This thesis is an investigation of the compilation methodology and of the properties of the synthesized circuits.

3 Application-Specific Hardware and Spatial Computation

From an architectural point of view Spatial Computation exhibits some desirable attributes:

- it is *program-specific*, and thus it has no interpretation overhead,

- it can exploit virtually unlimited instruction-level parallelism (ILP),
- it features mostly short and fast point-to-point wires driven by a single writer each (i.e., wires that require no arbitration),
- it is asynchronous, and thus latency tolerant; this makes it correct by design, even when it employs global structures, such as the memory access network,
- as in superscalar processors and dataflow machines, its computation is dynamically scheduled, based on the availability of data,
- it requires no centralized control structures,
- it has a modular structure;
- it is composed of *lean* hardware; in particular, both datapath and control-path contain no broadcast structures, no arbitration, no multi-ported memory or register files, no content-addressable structures, and would thus be able to run at a very high speed[1],
- it is easy to reason about; we believe it is thus easily amenable to formal verification.

In this thesis we investigate in detail a particular instance of Spatial Computation, with the following features:

- Each source-level procedure is translated into a distinct hardware structure;
- Each source-level operation is synthesized as a different hardware arithmetic unit; our circuits thus exhibit *no computational resource sharing*;
- All program data is stored in a single monolithic memory, accessible through a conventional memory hierarchy, including a load-store queue and caches;
- The synthesized hardware is an application-specific *static dataflow machine*. This means that (1) operations are executed based on the availability of data and (2) at any point in time there can exist in the circuit at most one result for each operation[2].

We define Application-Specific Hardware, abbreviated ASH, as the hardware structure implementing a particular program, synthesized under compiler control from the application source code; the translation is illustrated in Figure 3.

3.1 ASH and Computer Architecture

The features of Spatial Computation suggest that it may provide very good performance on data-intensive programs, which have high amounts of instruction-level parallelism. Indeed, as we show in this thesis, ASH can surpass by a comfortable margin a 4-wide out-of-order superscalar on media processing kernels.

However, as our analysis shows, some of the strengths of ASH are also its weaknesses when we consider control-intensive programs. The lack of branch prediction in ASH is a severe handicap for benchmarks

[1] The memory access network however requires some complex structures.

[2] In contrast, in dynamic dataflow [3], at some point in time there can be an arbitrary number of results "live" for each operation.

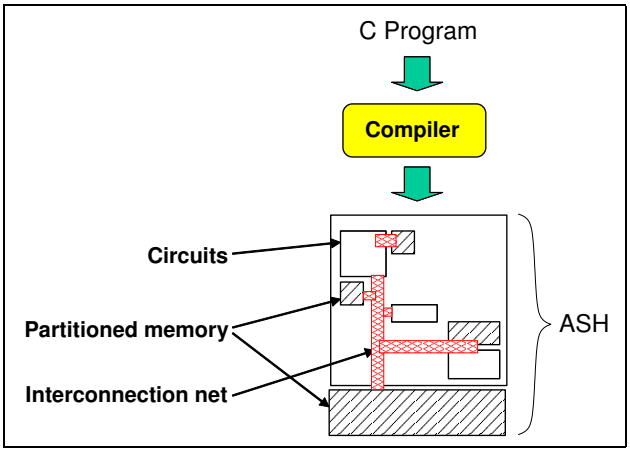

Figure 3: *Application-Specific Hardware is generated by the automatic synthesis of computational structures, memories and an interconnection network from C programs.*

where the ILP is low and the computation of the branch conditions is on the critical path. The distributed nature of ASH makes reaching memory expensive, and squashing speculation difficult.

These behaviors suggest that a versatile computer system should combine the strengths of both out-of-order processors, for control-intensive code, and Spatial Computation, for data-intensive kernels. This research opens the exploration for the detailed architecture of such a system, and mostly of a low-overhead hardware substrate suitable for Spatial Computation (when compared to contemporary FPGA devices).

4 Contributions

To our knowledge the following are original research contributions of this work:

Predicated SSA dataflow internal representation: we are the first to report on the wide scale usage of a compiler internal program representation that brings together static single-assignment, predication, forward speculation and a functional representation of side-effects. Several features of this representation make compiler development particularly efficient: (1) it has a precise semantics, (2) enables a succinct expression of most common program optimizations (3) it enables efficient reasoning about memory state and thus enables powerful memory optimizations, (4) it makes dataflow analyses simple and efficient.

SIDE as a new framework for dataflow analysis: our new hybrid dataflow analysis framework, Static Instantiation — Dynamic Evaluation uses code to dynamically evaluate dataflow information when a static evaluation is too conservative.

New compiler algorithms. We describe new optimization algorithms for redundancy elimination in memory accesses: register promotion and partial redundancy elimination for memory. We also describe algorithms for enhancing the pipelining of loops, such as a scheme for pipeline balancing, memory access pipelining in loops through fine-grained synchronization and loop decoupling.

Language extension: we have studied the efficiency of user annotations in C programs (through the use of `#pragma` statements) to convey pointer non-aliasing information to the compiler.

Dataflow model extensions: we have realized the first translation of the complete C language to a dataflow machine. This effort has crystallized a methodology for implementing imperative languages, including unstructured flow-of-control, recursion, side-effects and modifiable storage. We have incorporated predication and speculation in the dataflow model of execution, and we have suggested a way to perform the equivalent of branch prediction in dataflow machines. Other contributions to the dataflow model of execution are the use of lenient operations and an enhanced form of pipeline balancing.

Hardware synthesis: we have built the first complete system able to translate arbitrary C programs into hardware. This tool differs from most prior approaches in its change of perspective: C is not regarded as a hardware-description language, destined to describe an independently-executing piece of hardware; instead, the goal of the synthesis system is to directly implement a given application as a hardware circuit.

Asynchronous circuit design: we have provided the first tool that can be used to translate C programs into asynchronous circuits. The translation methodology is general enough to be applicable to any imperative language.

Embedded systems construction: we have built the CASH compiler, which can be used as a tool for very fast prototyping of embedded systems hardware accelerators and has shown its effectiveness on a wide range of media processing applications. Although not detailed in this document, we have suggested a new way of performing hardware-software partitioning, which relies on separating the policy from the mechanism; this partitioning is easily amenable to automatization. CASH together with the partitioning methodology constitute a very efficient method of exploring the design of complex application-specific system-on-a-chip devices using only the application source code.

New perspective on superscalars: we have performed a new limit study on the subject of instruction-level parallelism and the effectiveness of architectural features for exploiting it. Namely, by contrasting a static dataflow model with unbounded resources (ASH) against a superscalar processor implementation we have highlighted the effectiveness of a monolithic architecture at performing memory accesses, prediction and speculation and emulating a full dynamic dataflow model of execution.

5 CASH: A Compiler for Application-Specific Hardware

In this section we provide and overview of CASH, the compiler translating C programs into Application-Specific Hardware (ASH).

Figure 4 shows CASH within its environment. The inputs to the compiler are programs written in ANSI C. CASH represents the input program using a dataflow internal representation called Pegasus. The output of CASH is a custom hardware dataflow machine, which directly executes the input program. The result can be implemented either as a custom hardware device, or on a reconfigurable hardware substrate.

CASH translates the input program into a flat computational structure, which contains no interpretation engine. Roughly, to each operation in the source corresponds one arithmetic unit in the output circuit. This kind of computation structure is called Spatial Computation. It exhibits maximum parallelism and minimal resource sharing. The only resource sharing is at the procedure level, since different calls to the same procedure re-use the same circuit.

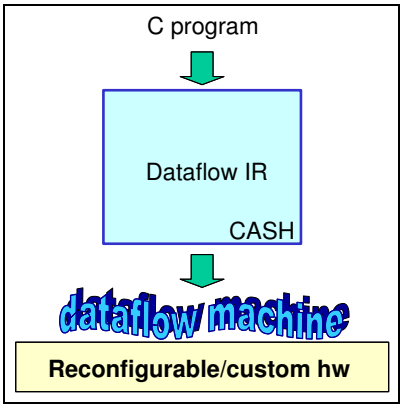

Figure 4: *The CASH compiler translates C programs into hardware.*

5.1 What CASH Does

With regard to the range of program transformations performed, CASH is much closer to a software compiler than to a CAD tool. We have implemented in CASH most textbook scalar optimizations; in addition, CASH has proven to be a great vehicle for implementing some memory-related optimizations, such as redundancy elimination and register promotion.

Most optimizations in CASH are applied at the level of hyperblocks, which tend to be relatively coarse program regions. Optimizations are thus, as effectiveness goes, somewhere between purely local, basic-block-based optimizations, and global, whole-procedure optimizations. An important point is that most innermost loops are hyperblocks, and thus are treated as an optimization unit.

The front-end performs procedure inlining and loop unrolling. The CASH core performs global constant propagation, constant folding, partial redundancy elimination for both scalars and memory, a wide range of algebraic optimizations, loop-invariant code motion, scalar promotion based on powerful memory disambiguation, strength reduction, unreachable- and dead code removal and Boolean simplifications.

The compiler also performs pipeline balancing for enhancing the throughput of the circuits which exhibit dataflow software pipelining.

6 Pegasus: an Internal Representation

Pegasus is the internal representation of CASH. Probably the most important feature of Pegasus is its *completeness*: the representation is self-contained, enabling a complete synthesis of the circuits, without requiring further information (internal representations with this property are called *executable*).

Three important computational paradigms are brought together in Pegasus: (1) executable intermediate representations, (2) dataflow machines and (3) asynchronous hardware circuits. Indeed, the most natural way to implement a custom dataflow machine (i.e., one executing a fixed program) is by using asynchronous hardware. Both the dataflow machine operations and the asynchronous hardware computational units start computing when they receive their input operands, and both use fine-grained synchronization between data producers and consumers to indicate the availability and consumption of data. Pegasus is original because it is used for compiling an imperative language into dataflow machines. The overwhelming majority of the work in compilation for dataflow machine dealt with functional, single-assignment languages. On the other hand, asynchronous circuits are customarily described in some form of Communicating Sequential Processes. Both these classes of languages are very different in nature from C.

Certainly, the adequacy of dataflow intermediate forms for compiling imperative language has been noted before; in fact, many of the Pegasus features are inspired by Dependence Flow Graphs [8], which is a dataflow representation used to represent FORTRAN programs. The key technique allowing us to bridge the semantic gap between imperative languages and asynchronous dataflow is Static Single Assignment (SSA) [6]. SSA is an intermediate representation used for compiling imperative programs in which each variable is assigned only once. As such it can be seen as a functional program [2]. Indeed, Pegasus represents the scalar part of the computation of C programs as a functional, static-single assignment representation.

The **main contribution** of this work in the domain of compiler intermediate representations is the seamless extension of SSA to incorporate other modern features of compiler representations, such as a representation of memory dependences, predication and (forward) speculation. While other intermediate program representations have each previously unified some of these aspects, we believe we are the first to bring all of them together in a coherent, semantically precise representation.

Pegasus integrates several important compiler technologies, drawing on their strengths:

Dependence representation: Pegasus is a form of program-dependence graph, summarizing data-flow, data dependence and control-flow information. Pegasus makes explicit *all* the dependences between operations (either scalar computations or memory operations).

Soundness: as any representation exposing parallelism, Pegasus is non-deterministic (i.e., it does not enforce a total execution order of the operations). However, it can be easily proved that Pegasus has the Church-Rosser property (i.e., confluence): the final result of the computation is the same, for any legal execution order of the operations.

Precision: Pegasus' building blocks have a precise semantics; the semantics is compositional. Moreover, a Pegasus representation is a complete executable form, which describes unambiguously the meaning of a program, without the need for external annotations.

Single-Assignment: Pegasus is a sparse intermediate representation, based on a (slightly modified) form of static-single assignment. As such, it shares with SSA the space and time asymptotic complexity, enabling fast and efficient compiler manipulations. Pegasus uses classical SSA to represent value flow within each hyperblock; the modification consists in the treatment of the inter-hyperblock flow: we place a merge operator (ϕ node) at each hyperblock entry point for each variable live at that point. While this modification theoretically has worse asymptotic properties, in practice it is completely acceptable.

Expressive: Pegasus enables very compact implementations for a large number of important program optimizations. In fact, most of the common scalar optimizations can be rephrased as simple term-rewriting rules operating on Pegasus graphs; implementing some of these optimizations on CFGs requires the computation of dataflow information[3]. This fact is important not because it saves the compiler the time to perform the dataflow analysis itself (for most of the optimizations only simple bit-vector, or other constant-depth lattice analyses are required); the importance is in saving the effort to *update* the dataflow information when program transformations are applied. Although research has shown how to perform incremental dataflow updates, maintaining several pieces of dataflow information while performing complex program transformations is a daunting compiler-engineering task.

[3]Unfortunately the term *dataflow* is overloaded; in this document it is used to denote both dataflow machines and dataflow analyses. We hope that the context is always clear enough to disambiguate its meaning.

Verifiable: Due to the compositionality of the semantics, the correctness of many optimization can be easily argued (especially the ones performing term-rewriting): if a program fragment is replaced by an equivalent one, the end-to-end program semantics does not change. An important piece of future work is to apply formal verification techniques for proving that Pegasus transformations preserve program semantics.

Compositionality also simplifies the presentation of the examples in this text: we can unambiguously isolate contiguous subgraphs of a larger code fragment independent on their environment, since the semantics of the subgraph is well-defined.

Predication and speculation: are core constructs in Pegasus. The former is used for translating control-flow constructs into data-flow, and the latter for reducing the criticality of the control-dependences. Both these transformations are borrowed in their form used in the compilation for EPIC architectures. They are effective in boosting the exposed instruction-level parallelism. Both predication and speculation are employed in Pegasus at the hyperblock level; speculation is thus only forward speculation, which evaluates all outcomes of forward branches within a hyperblock.

7 Optimizations

The fourth chapter of the thesis discusses in detail the optimizations performed by CASH. We make several research contributions to the subject of compiler optimizations:

- We present several new, more powerful versions of classical optimizations. Some of them are applicable in the context of traditional compilers as well, while other are more intimately tied to the features of the Pegasus internal representation.

- We suggest a new class of optimizations, based on *dynamic dataflow analysis*. As an example we present a new, optimal algorithm for register promotion, which handles inter-iteration promotion in the presence of control-flow. The idea of these optimizations is, instead of using static conservative approximations to dataflow facts, to *instantiate the dataflow computation at run-time* and thus to obtain dynamic optimality.

- We show that Pegasus can uniformly handle not only scalar optimizations, but also optimizations involving memory. Other efficient program representations treat memory as a second-order object, resorting to external annotations and analyses in order to handle efficiently side-effect analysis. For example, while SSA is an IR that is widely recognized as enabling many efficient and powerful optimizations it cannot be easily applied in the presence of pointers. (Several schemes have been proposed, but we believe that none of them fits as naturally as Pegasus' in the framework of single-assignment.) We show how Pegasus enables efficient and powerful optimizations in the presence of pointers.

- We show how Pegasus not only handles optimizations naturally in the presence of predication and speculation (most classical optimizations and dataflow analyses break-down in the presence of predication), but even takes advantage of predicates in order to implement very elegant and natural program transformations, without giving up any of the desirable features of SSA. We show how to use predication and SSA together for memory code hoisting (subsuming partial redundancy elimination and global common-subexpression elimination for memory operation), removing redundant loads, loads after stores, and dead stores.

- Unlike other intermediate representations, Pegasus represents essential information in a program without the need for external annotations. Moreover, the fact that Pegasus directly connects definers and users of values makes unnecessary many traditional dataflow analyses: they are summarized by the edges. Thus, many optimizations are reduced to a local rewriting of the graph. This is important not because of the elimination of the initial dataflow computation, but because *dataflow information is maintained automatically in the presence of program transformations*. There is thus no need for cumbersome (incremental or complete) re-computations of dataflow facts after optimizations.

- As a byproduct of the sparsity of the representation and of the lack of need of dataflow analyses many optimizations are linear time, either worst-case or in practice. This enabled us to use a very aggressive optimization schedule, by iterating most optimizations until convergence. We believe that CASH is unique in this respect; all compilers that we know of have a relatively fixed schedule of optimizations, while in CASH some optimizations may be applied many times.

- The sparsity of Pegasus and its "def-use" nature makes the implementation of dataflow analyses practically trivial. We illustrate this by describing a general dataflow framework and showing how several analyses fit into it in.

- We introduce *loop decoupling*, a new loop parallelization algorithm for handling memory dependences, which relies on creating a computation structure that *dynamically controls the slip* between two parallel computations.

- We bring new insights into the implementation of software pipelining. Our approach, inspired by the paradigm of dataflow software pipelining, was devised for implementation in Spatial Computation. Our approach is immediately applicable on multithreaded processors, and is much simpler than traditional software pipelining, while being also tolerant of unpredictable latency events.

8 ASH at Run-time

8.1 Lenient Execution

Pegasus uses explicit multiplexors to represent the join of multiple definitions of a variable. Hyperblocks are the basic unit of (speculative) execution: all operations within a hyperblock are executed to completion every time the execution reaches that hyperblock.

In the literature about EPIC processors, a well-known problem of speculation is that the execution time on the multiple control-flow paths may be different [4]. Figure 5 illustrates the problem of balancing: the critical path of the entire construct is the longest (worst-case) of the critical paths of the combined conditional paths. Traditional software solutions use either profiling, to ensure that the long path is overwhelmingly often selected at run-time, or exclude certain hyperblock topologies from consideration, rejecting the predication of paths which differ widely in length.

In Spatial Computation we can use a completely different method to solve this problem: we make multiplexors and predicate computation operations *lenient*. Leniency is a property of the execution of an operation, related to strictness and laziness. An operation is strict if it requires all its inputs to compute the output. An operation is non-strict if it can produce a result with just some of the inputs. Lenient operators generate a result as soon as possible, while still expecting all inputs to eventually become available. For example, a "logical and" operation can determine that the output is "false" as soon as one of its inputs is

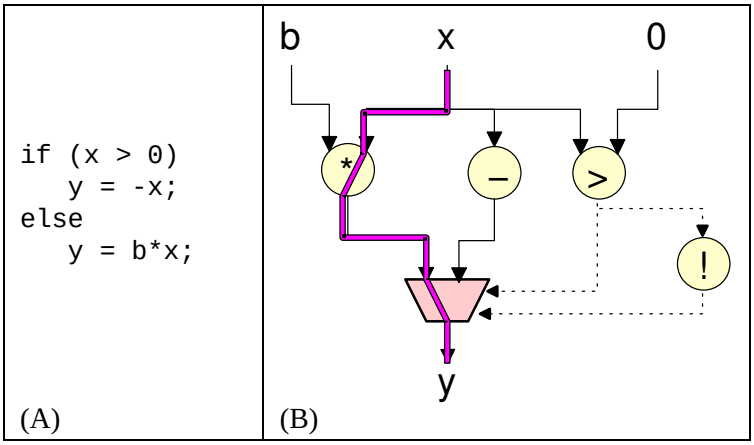

Figure 5: *(A) Sample program fragment and (B) the corresponding Pegasus circuit with the static critical path highlighted.*

"false". In our implementation lenient operations however defer the sending of acknowledgments until all of the inputs have been received.

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 "false", it also evaluates the right one. A lenient evaluation of "and" however generates a "false" result as soon as *either* input is known to be "false". Short-circuit evaluation is a static property, while lenient execution is a dynamic property.

Multiplexors have lenient implementations as well: as soon as a selector is "true" and the corresponding data is available, a mux generates the output. As a result of leniency, at least for this sample circuit, *the dynamic critical path is the same as in a non-speculative implementation.* I.e., if the multiplication output is not used, it does not affect the critical path. The multiplication delay can still impact the dynamic critical path if it is within a loop, but this can be managed by pipelining the implementation of the multiplier.

Besides Boolean operations and multiplexors, there are other operations which are only "partially" lenient. For example, 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 (by raising the "data ready" signal), since the actual output is irrelevant. Other operations could be made lenient in some weaker sense still; for example, a multiplication receiving a 0 input could immediately output a 0 result.

8.2 Dataflow Software Pipelining

One of the most interesting properties of Spatial Computation is its automatic exploitation of pipeline parallelism. This phenomenon is a consequence of the dataflow nature of ASH, and has been well studied in the dataflow literature. In the case of static dataflow machines (i.e., having at most one data item on each edge) this phenomenon is called dataflow software pipelining [10]. The name suggests the close relationship between this form of pipelining and the traditional software pipelining optimization, customarily used in VLIW processors to speed-up scientific code.

We use the simple program in Figure 6 to illustrate this phenomenon. This program uses `i` as a basic induction variable to iterate from 1 to 10, and `sum` to accumulate the sum of the squares of `i`.

Let us further assume that the multiplier implementation is pipelined with 8 stages. In Figure 7 we show the snapshots for a few consecutive "clock ticks" of this circuit.

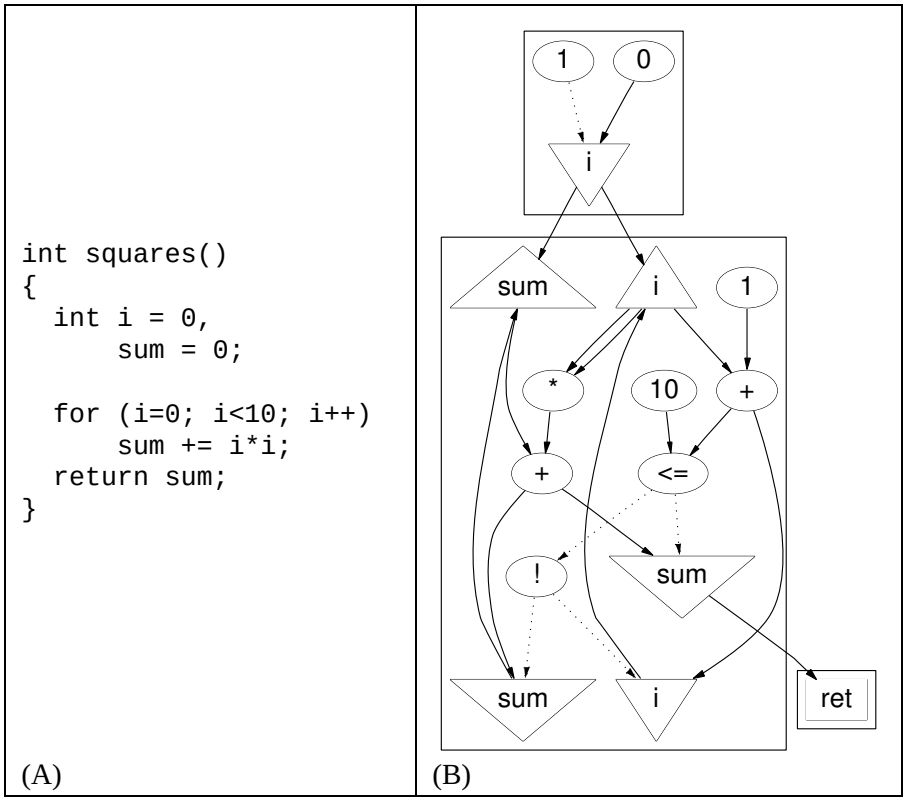

Figure 6: *Program exhibiting dataflow software pipelining.*

Notice that the original CFG loop has been decomposed into two independent dataflow loops, one computing the value of `i` and the other computing the value of `sum`. The two loops are connected by the multiplier, which gets its input from one loop and sends its output to the second loop. At run-time the `i` loop is, in the last snapshot in Figure 7, more than one iteration ahead of the `sum` loop.

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`.

Let us notice however some important differences between the two types of software pipelining:

- Traditional software pipelining relies on precise scheduling of the operations for each clock cycle; unpredictable events and unknown latency operations (such as cache misses) do not fit well within the framework. In contrast, Spatial Computation requires no scheduling and automatically adjusts when the schedule is disrupted, because the execution of all operations is completely decoupled.

- Effective software pipelining relies on accurate resource usage modeling by each instruction, some of which are very hard to reason about, such as cache bank conflicts. In Spatial Computation as many resources as necessary can be synthesized; unpredictable contention for the shared resources, such as LSQ ports or cache banks has only local effects on the schedule.

- Sophisticated production-quality implementations of software pipelining require a huge amount of code; a MIPS R8000 implementation described in the literature has 33,000 lines of code, which is more than the complete CASH core.

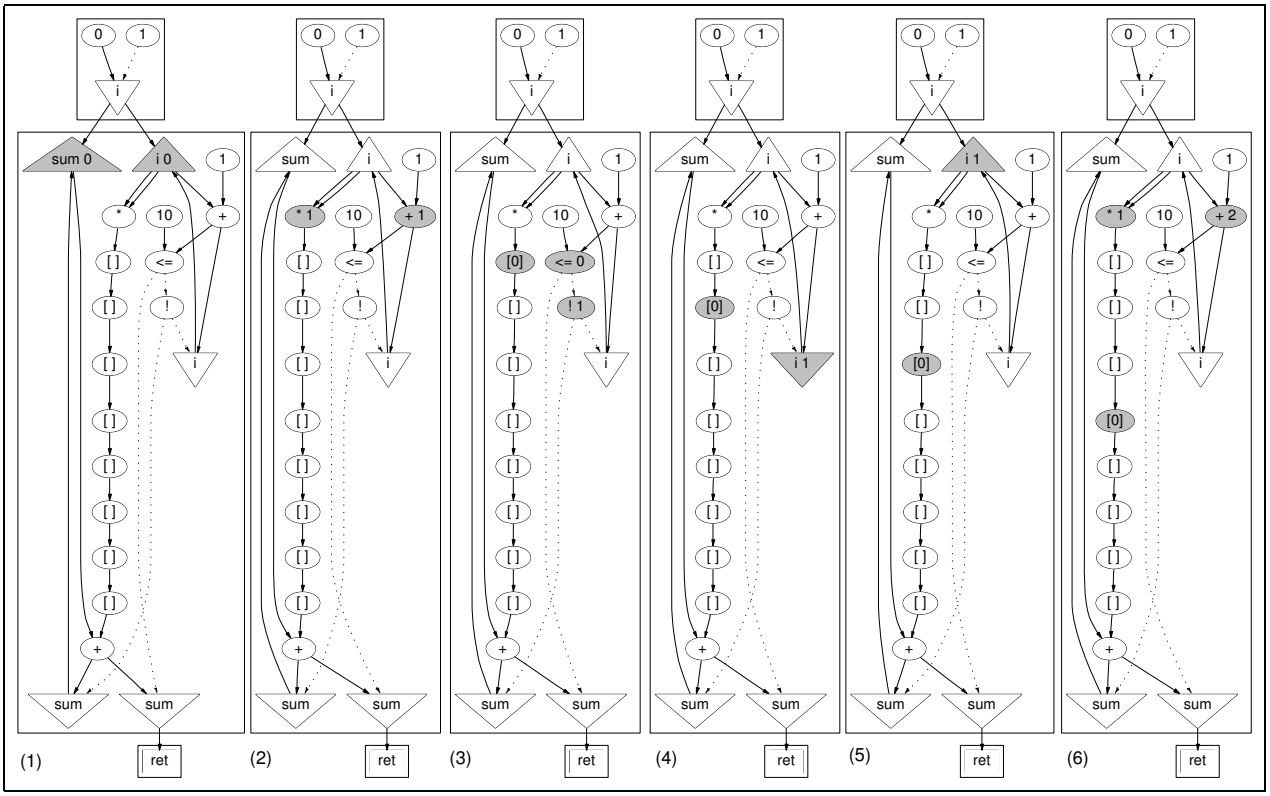

Figure 7: *Snapshots of the execution of the circuit in Figure 6, assuming that the multiplier implementation is pipelined with 8 stages, and that the negation latency is 0. In the last snapshot two different values of* `i` *are simultaneously present in the multiplier pipeline.*

9 Implementing Media Kernels in ASH

In this section we evaluate the effectiveness of CASH on selected kernels from embedded-type benchmarks. We use programs from the Mediabench [9] benchmark suite. For selecting the kernels we first run each benchmark on a superscalar processor simulator and collect profiling information. Next we compile each kernel using CASH and the rest of the benchmark using gcc, and we simulate the resulting "executable" by measuring only the kernel execution. The baseline performance is measured using the SimpleScalar 3.0 simulator [5], configured as a 4-way out-of-order superscalar.

On the ASH system the load-store queue has more input ports, due to the high bandwidth requirements from the memory access protocol. The L1 and L2 data caches, and the memory latency are all identical on the compared systems.

Most of the operation latencies are assumed to be identical in ASH and a prototypical processor, which will be the basis for our comparison. Some values are smaller for ASH, due to its specialized hardware nature; for example, we assume that constant shifts or negation operations can be made in 0 cycles. The assumption for negation is justified because it is always a 1-bit operation, and we assume that in a hardware implementation, the source of the negation operation always computes not only the correct value but also the complement. We assume that the timing of most arithmetic operations is the same on both systems, although this probably is unfair to the processor, since its arithmetic units are highly optimized, trading-off area for performance.

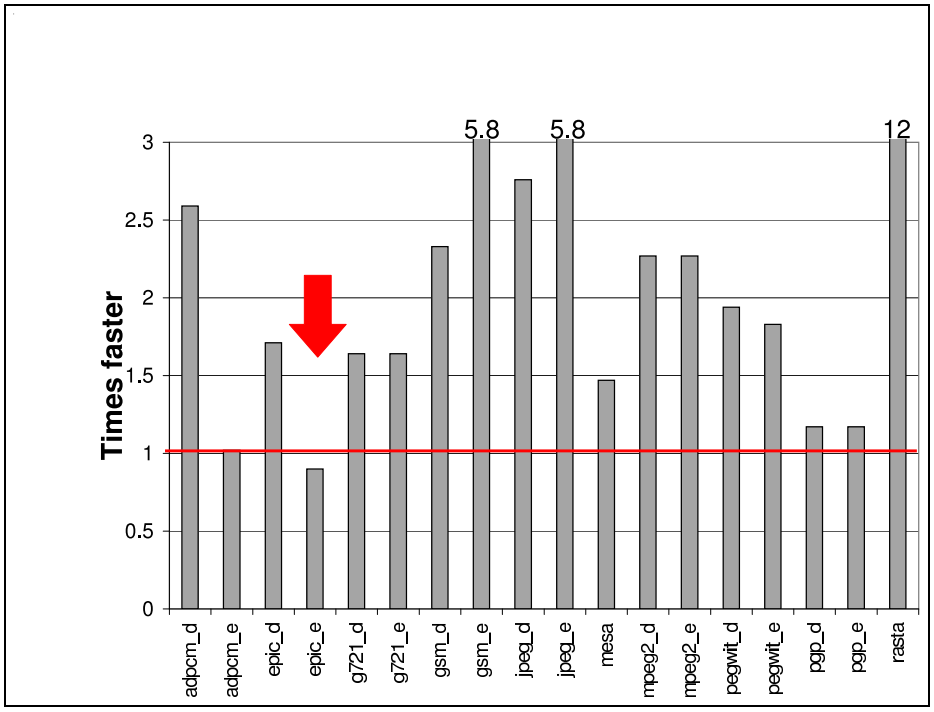

Figure 8: *Speed-up of ASH versus a 4-way out-of-order superscalar processor. The only kernel exhibiting a slowdown is* `epic_e`.

In CASH we use very aggressive loop unrolling, and we disable inlining, unless otherwise specified. We use the full complement of the optimizations of the compilers for both systems.

Note that since ASH has no instruction issue/decode/dispatch logic, it cannot be limited by instruction processing. We use rather optimistic values for the instruction cache for the processor in order not to have an artificial disadvantage.

9.1 Performance Results

We express performance in machine cycles for executing the kernel alone. We time the kernel on the processor and on ASH. We implicitly assume the same "clock" for both implementations.

Figure 8 shows how much faster ASH is compared to the superscalar core. Except one program, `epic_e` which slows down, and one program `adpcm_e`, which breaks-even (with a speed-up of only 2%), all kernels exhibit significant speed-ups, ranging from 50% up to 1200% for `rasta`.

Figure 9 shows the indisputable advantage of Spatial Computation: unlimited parallelism, exploited post-hoc. It shows the sustained IPC for both the processor and ASH for the selected kernels. On all kernels ASH sustains a higher IPC, and, except `adpcm_e`, an IPC higher than the maximum theoretically available from the processor. In fact, excepting three benchmarks (`adpcm_e` and both `g721`) the IPC of ASH is more than twice that of the processor.

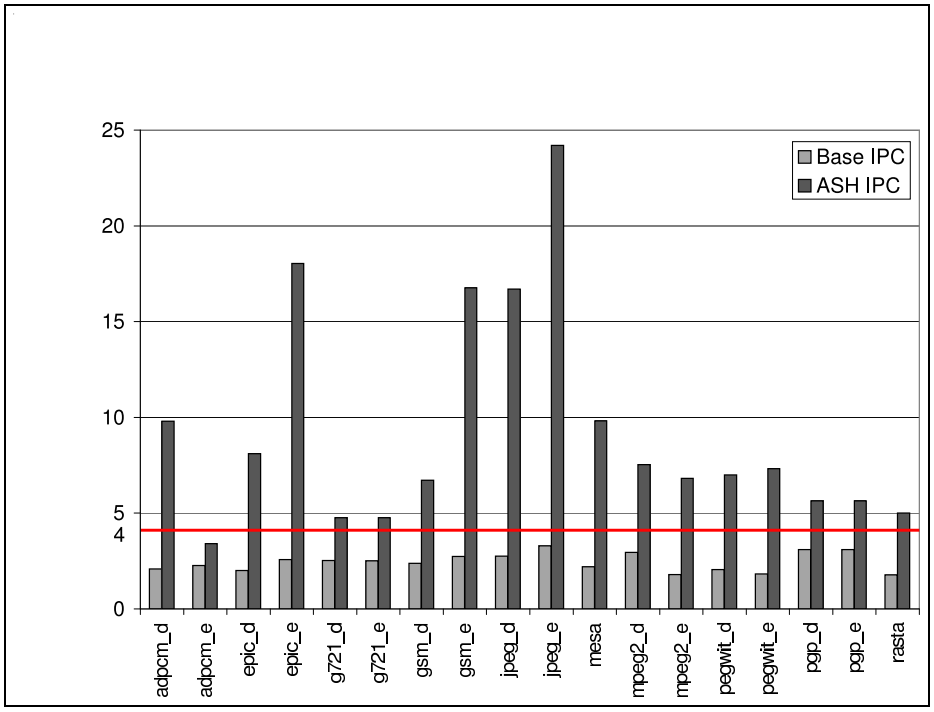

Figure 9: *Average IPC (Instructions Per Cycle) for superscalar and ASH. The upper limit (4) for the processor is indicated.*

10 ASH Versus SuperScalar

In this section we compile whole programs using CASH and evaluate their potential performance. We compare to a superscalar processor baseline. We assume the same operation latencies as in the comparison of media kernels.

Some of these assumptions are clearly optimistic for ASH when dealing with whole programs. In particular, the complexity of the memory access network and of the interconnection medium for procedure calls and returns cannot be assumed constant any more. In the best case the latency of these networks should grow as $\sqrt{n}$, where n is the size of the compiled program, assuming a two-dimensional layout. One should thus interpret the data in this section more as a limit study than as absolute magnitudes.

10.1 Performance Measurements

Figure 10 shows the speed-up of complete applications from SpecInt95 executed under ASH when compared to a superscalar processor. Unfortunately most of the numbers indicate a slowdown. Only `099.go`, which is very array-intensive, and `132.ijpeg`, which is very similar to the Mediabench `jpeg` programs, show some performance improvements. Results for SpecInt2K show similar trends, and are not presented due to excessive simulation time requirements.

10.2 Analysis of Performance Results

The thesis presents a detailed analysis of the influence of architectural features on the performance results. The following (lack) of features penalizes ASH:

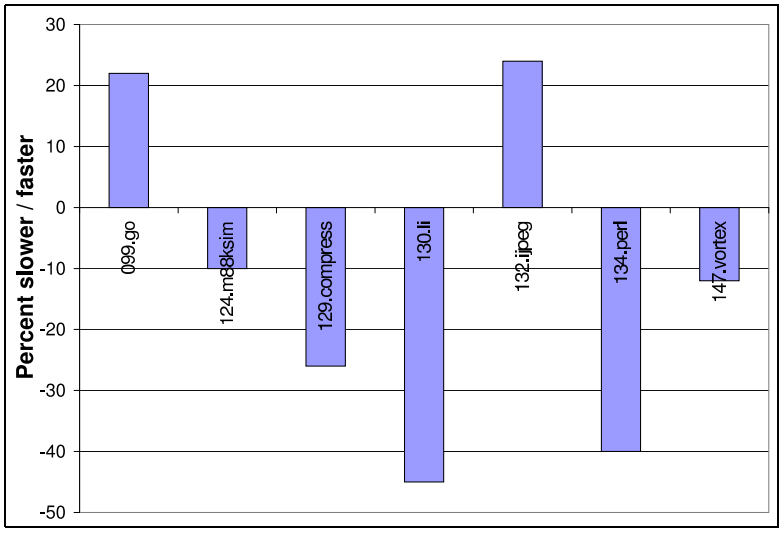

Figure 10: *Speed-up of SpecInt95 benchmarks executed on ASH. The baseline is a superscalar 4-way out-of-order processor. Negative values indicate a slow-down.*

- The lack of branch prediction, which may introduce control dependences on the critical path;
- The strictness (non-leniency) of "call" and "return" operations;
- Insufficient optimization of Boolean computations;
- Excessive register save/restore overhead for implementing recursive calls;
- The fact that "joins" in the control-flow graph are translated into non-free operations (multiplexors and merges);
- The lack of register renaming in ASH creates additional output-dependences;
- Accesses to memory are "remote" and thus require longer latencies.

11 Conclusions

Traditional computer architectures are clearly segmented by the Instruction Set Architecture (ISA), which the hardware-software interface definition: compilers and interpreters sit "above", while the processor hardware is "below", functioning as an interpreter of the instruction set. In this thesis we have investigated a computational structure which foregoes the existence of an ISA altogether. In this setup the compiler *is* the architecture.

We have investigated one particular instance of an ISA-less architecture, which we dubbed "Spatial Computation". In Spatial Computation the program is translated directly into an executable hardware form, without using an 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 computation engine is completely distributed, featuring no global communication or broadcast, no global register files, no associative structures, and using

resource arbitration only for accessing the global monolithic 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 based on availability of data and not according to a fixed schedule. A natural vehicle for implementing self-scheduled computations is provided by asynchronous hardware.

This thesis has explored both the compile-time and run-time properties of such an architecture. The main contribution of this work with respect to the compilation methodology has been the introduction of a *dataflow intermediate representation* which seamlessly embeds several modern compilation technologies, such as predication, speculation, static-single assignment, an explicit representation of memory dependences, and a precise semantics. We have shown how many classical and several novel optimizations can be efficiently implemented using this representation.

Our investigation in the run-time aspects of Spatial Computation has shown that it indeed can offer superior performance when used to implement programs with resource requirements substantially exceeding the ones available in a traditional CPU. In particular, the execution of multimedia program kernels can achieve very high performance in Spatial Computation, at the expense of very little power. A detailed investigation of the run-time behavior of the spatial structures has also shown that their distributed nature forces them to incur small, but non-negligible overheads when handling control-intensive programs. We have shown that the use of global structures (such as branch prediction, control speculation and register renaming) in traditional superscalar processors can thus more efficiently execute control-intensive code.

Since Spatial Computation 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.

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] Andrew W. Appel. SSA is functional programming. ACM SIGPLAN Notices v. 33, no. 4, April 1998.

[3] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3), 1990. Also as MIT Computations Structures Group Technical Memo Memo-271, 1987.

[4] 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.

[5] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25 (3), pages 13–25. ACM SIGARCH, June 1997.

[6] 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.

[7] Michael Flynn. Computer architecture and technology: some thoughts on the road ahead. http://www.acm.org/sigs/conferences/fcrc/PlenaryTalks/Flynn.pdf, June 2003. Presentation at the FCRC plenary talks.

[8] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 78–89, Albuquerque, New Mexico, June 1993.

[9] 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.

[10] Gao Guang Rong. An implementation scheme for array operations in static data flow computers. Technical Report MIT-LCS-TR-280, MIT LCS, May 1982.