# Achieving Superscalar Performance without Superscalar Overheads – A Dataflow Compiler IR for Custom Computing

**Ali Mustafa Zaidi and David J. Greaves**

**University of Cambridge Computer Laboratory**
**Cambridge CB3 0FD, UK**
{ali-mustafa.zaidi, david.greaves}@cl.cam.ac.uk

**Abstract**

The difficulty of effectively parallelizing code for multicore processors, combined with the end of threshold voltage scaling has resulted in the problem of 'Dark Silicon', severely limiting performance scaling despite Moore's Law. To address dark silicon, not only must we drastically improve the energy efficiency of computation, but due to Amdahl's Law, we must do so without compromising sequential performance. Designers increasingly utilize custom hardware to dramatically improve both efficiency and performance in increasingly heterogeneous architectures. Unfortunately, while it efficiently accelerates numeric, data-parallel applications, custom hardware often exhibits poor performance on sequential code, so complex, power-hungry superscalar processors must still be utilized. This paper addresses the problem of improving sequential performance in custom hardware by (a) switching from a statically scheduled to a dynamically scheduled (dataflow) execution model, and (b) developing a new compiler IR for high-level synthesis that enables aggressive exposition of ILP even in the presence of complex control flow. This new IR is directly implemented as a static dataflow graph in hardware by our high-level synthesis tool-chain, and shows an average speedup of 1.13× over equivalent hardware generated using LegUp, an existing HLS tool. In addition, our new IR allows us to further trade area & energy for performance, increasing the average speedup to 1.55×, through loop unrolling, with a peak speedup of 4.05×. Our custom hardware is able to approach the sequential cycle-counts of an Intel Nehalem Core i7 superscalar processor, while consuming on average only 0.25× the energy of an in-order Altera Nios IIf processor.

**1998 ACM Subject Classification** B.5.1 Design, B.6.1 Design Styles, B.6.3 Design Aids, C.1.3 Other Architecture Styles, D.3.2 Language Classifications, D.3.4 Processors

**Keywords and phrases** High-level Synthesis, Instruction Level Parallelism, Custom Computing, Compilers, Dark Silicon

**Digital Object Identifier** 10.4230/OASIcs.ICCSW.2013.136

## 1 Introduction

Despite ongoing exponential growth of on-chip resources with Moore's Law, the performance scalability of future designs will be increasingly restricted. This is because the total *usable* on-chip resources will be growing at a much slower rate, due to the recently identified problem of Dark Silicon [6, 7]. Esmaelzadeh et al. identify two main sources of Dark Silicon:

1. *Amdahl's Law*: With the exception of certain numeric, data-parallel applications, most applications in the general-purpose domain lack sufficient *explicit* parallelism to make full use of the ever increasing number of cores on chip [1]. Due to Amdahl's Law, the overall speed-up for such applications is strictly constrained by sequential performance.

2013 Imperial College Computing Student Workshop (ICCSW'13).
Editors: Andew V. Jones, Nicholas Ng; pp. 136–144

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2. *Utilization Wall*: Even when an embarrassingly parallel application is not limited by Amdahl's Law, overall performance scaling will still be limited by the Utilization Wall [7, 17]: with each process generation, a decreasing fraction of on-chip transistor resources may be switched at full speed at one time, in order to meet the power budget.

Recent work has exploited custom (and reconfigurable) hardware to improve the per-core energy efficiency for the general-purpose application domain [17, 3]. Unfortunately, sequential code often exhibits much lower performance in custom hardware than a typical out-of-order superscalar processor [2, 3, 17]. For the general-purpose domain, sequential code involves irregular memory accesses and complex control-flow (e.g. nested loops, data-dependent branching). Currently the only means of achieving high performance on such code is through the use of complex, inefficient out-of-order superscalar processors. Such processors exhibit an exponential increase in power dissipation as performance increases [8].

This puts us between a rock and a hard place: without utilizing complex processors, performance scaling is limited by Amdahl's Law and poor sequential performance, but with such processors, the Utilization Wall limits speed-up by limiting the total active resources at any time. Esmaelzadeh et al. [6] focused on desktop, server and workstation domains, that have a reasonable power budget of 20-200W. This problem is exacerbated even further when we consider the increasingly important and rapidly growing portable computing domain, where power budgets are further limited to only 0.5-5W.

Our goal is to enable much more pervasive utilization of custom or reconfigurable hardware for general-purpose computation in order to mitigate the effects of dark silicon. For this, we must (a) overcome the performance limitations on sequential code in custom hardware, (b) without compromising its inherent energy-efficiency, while (c) requiring minimal programmer effort (i.e. minimal or no alterations to the source code or programming language).

To this end, we develop a new compiler intermediate representation (IR), called the Value State Flow Graph (VSFG), that exposes instruction-level parallelism (ILP) from sequential code even in the presence of complex control-flow. The VSFG is also designed to be directly implementable as custom hardware, replacing the traditionally used Control-Data Flow Graph (CDFG) [16]. To test our new IR, we have implemented a new high-level synthesis (HLS) tool-chain, that compiles from LLVM [13] to the VSFG, then implements it as a Verilog hardware description. Unlike the statically-scheduled execution model of traditional custom hardware, we employ the dynamically-scheduled 'Spatial Computation' model [3], in order to match the dynamic scheduling advantages of out-of-order processors, allowing for better tolerance of variable latencies and statically unpredictable behaviour.

## 2 Enhancing sequential performance in Custom Hardware

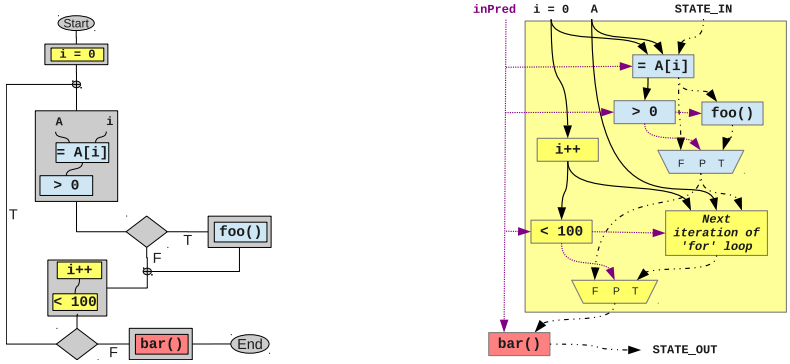
There are two main reasons that out-of-order superscalar processors are able to achieve higher performance on control-flow intensive sequential code [15]:

- Aggressive control-flow speculation to exploit ILP from across multiple basic-blocks, and
- Dynamic execution scheduling of instructions, approximating the dynamic dataflow execution model at runtime.

**The Superscalar performance advantage: A Case Study.** Listing 1 presents a code sample, and Figure 1a presents its equivalent CDFG (blue blocks represent '*if*' statement operations, while yellow blocks form the remainder of the *for* loop). Branch prediction in a superscalar processor will be able to overcome the control-flow dependences between the three basic blocks

**Listing 1** Example C Code

```
for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
bar();
```




**(a)** The CDFG for Listing 1.

**(b)** The VSFG for Listing 1.

**Figure 1** The equivalent CDFG and VSFG for the code given in Listing 1.

composing the for-loop. Furthermore, branch prediction enables *dynamic* unrollling of the for-loop, and exploitation ILP from across multiple iterations. Conventional processors make use of a centralized in-order commit buffer (i.e. *re-order buffer*) to selectively commit executed instructions in the correct CDFG order. In case of misprediction, executed instructions from mispredicted paths can simply be discarded, preserving correct program state.

On the other hand, custom hardware lacks the mechanisms to perform such aggressive control-flow speculation: unlike a conventional processor, there is no centralized commit buffer that can be used for misspeculation recovery. In the case of some forward (i.e. *if-else*) branches, it is sometimes possible to employ *if-conversion* to perform speculation by executing both sides of the branch and then discarding the result from the false path. However, this technique is limited to the simple cases where only data-flow is involved – the *if* branch in Listing 1 may invoke a separate function with its own control and data flow, and hence cannot simply be if-converted for speculation in hardware.

As it is not possible to perform speculation on backwards branches (i.e. loops) in custom hardware, High-level synthesis tools attempt to overcome this limitation by statically unrolling (as shown in Figure 2), flattening and pipelining loops in order to decrease the number of backwards branches that would be dynamically executed, but this can significantly increase the complexity of the centralized finite-state machines, resulting in very long combinational paths that can overwhelm any gains in ILP [11, 9].

In addition to aggressive control-flow speculation, superscalar processors employ dynamic execution scheduling, which helps in dealing with unpredictable behavior at runtime. Instructions are allowed to execute as soon as their input operands, as well as the appropriate execution resources, become available. Processors can even have multiple instances of the same instruction (say from a tightly wound loop) in flight, with their results written to different locations via register renaming. Using renaming, contemporary processors are able to approximate the dynamic-dataflow model of execution, allowing them to easily adapt to runtime variability to improve performance. For instance, even in the case that the load ("=

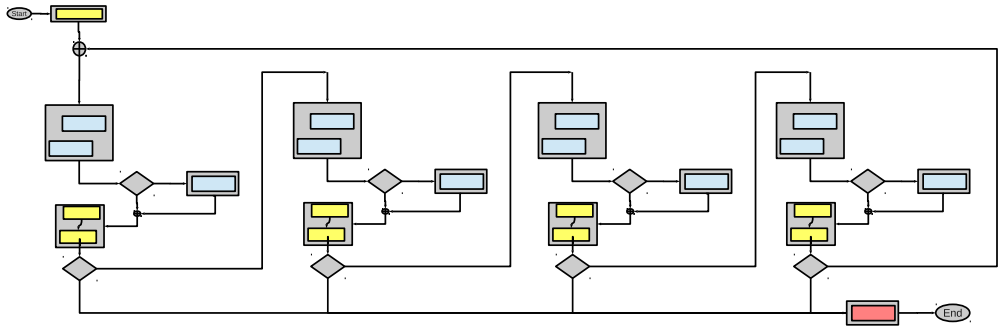



**Figure 2** The CDFG from Figure 1a with the loop unrolled 4 times.

*A[i]*") instruction encounters a cache miss, all the remaining operations that are in flight from the multiple blocks and loop iterations may still execute in dataflow order - only those operations that are dependent on the value of the load will be delayed.

On the other hand, custom hardware employs static execution scheduling, where the execution schedule for operations is determined at compile-time, and implemented at run-time by a centralized state machine. This means that such hardware can only be conservatively scheduled for the multiple possible control-flow paths through the code, leaving it unable to adapt to runtime variability that may occur due to data-dependent control-flow, variable-latency operations, or unpredictable events such as cache misses.

The combination of these factors results in custom hardware exhibiting poor performance when implementing general-purpose sequential code via high-level synthesis. Venkatesh et al [17] generate custom hardware *conservation cores* from hot regions of sequential general-purpose code to improve energy efficiency. However, their cores are at best only able to match the performance of an in-order MIPS 24KE processor.

**Matching Superscalar performance with the VSFG.** To overcome the sequential performance issues of custom hardware, we propose two key changes during high-level synthesis:

- Instead of using a static scheduling based execution model for custom hardware, a dynamically scheduled execution model like *Spatial Computation* should be used [3], that implements code as a static dataflow graph in hardware.
- A new compiler IR is needed to replace the CDFG based IRs that are traditionally used for hardware synthesis. This new IR should be based on the Value State Dependence Graph (VSDG) [14] as it has no explicit representations of control-flow, instead only representing the necessary value and state dependences in the program.

A new compiler IR for implementing spatial computation, called the Value State Flow Graph (VSFG), has been developed based on the VSDG but modified for direct implementation in hardware as a static dataflow machine. The VSFG for Listing 1 is shown in Figure 1b. Unlike the CDFG, there is no subdivision of operations into basic blocks, and consequently, no notion of *flow of control* from one block to another. Instead, only dataflow dependences are represented, along with a sequentializing state-edge (dashed line in Figure 1b) that ensures that all side-effecting operations occur in the correct program order.

Instead of flow of control, the execution of operations is controlled through the use of *predicates* (purple dotted line in Figure 1b) – boolean expressions generated based on the control-flow of the CDFG. The VSFG is also a hierarchical graph – all loops and function calls are represented as nested subgraphs. From the perspective of any level in the graph hierarchy,

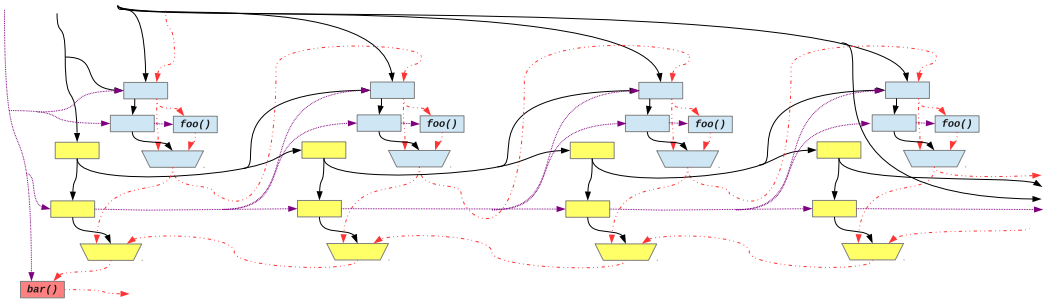



**Figure 3** The VSFG from Figure 1b with the loop unrolled 4 times.

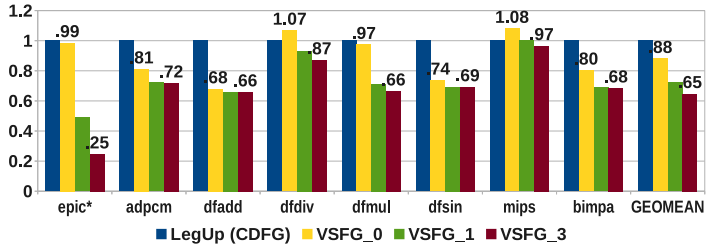
nested subgraphs appear as ordinary dataflow operations with their defined dataflow inputs and outputs (including predicate and state). The only difference being that nested subgraphs may exhibit a variable execution latency. As with other dataflow operations, multiple such subgraphs may execute concurrently, so long as their dataflow dependences are satisfied.

The VSFG is a directed acyclic graph, so loops are implemented without introducing explicit cycles in the graph by representing them instead as tail-recursive functions. As can be seen from Figure 1b, the *"Next iteration of the for-loop"* is represented as a nested subgraph in the same way as both the *foo()* and *bar()* functions. This presents a key advantage when we try to extract ILP from across multiple iterations of a loop. Any of the nested subgraphs in a VSFG can be *flattened* into the body of the parent graph. In the case of loops, flattening the subgraph representing the tail-recursive loop call is essentially equivalent to unrolling the loop. Figure 3 shows the for loop *unrolled* 4 times. Furthermore, as each loop is implemented within its own subgraph, this type of unrolling may be implemented within different subgraphs independently of others. Therefore, it is possible in the VSFG to exploit ILP by unrolling each loop within a loop nest independently of the other loops. (Note that in the actual hardware implementation, cycles must be reintroduced once the appropriate degree of unrolling has been done for each loop).

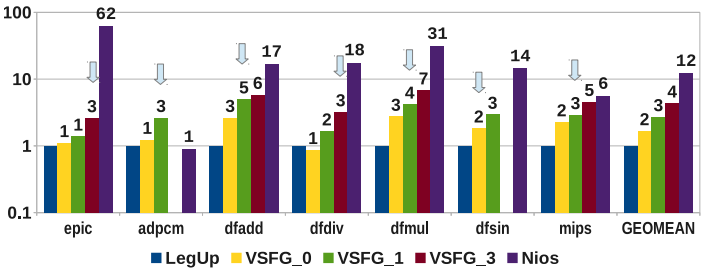
Thanks to the explicit control predicates, is also possible to perform aggressive control flow speculation by selectively controlling the execution of subgraphs. For instance, for the *foo()* function subgraph, we may choose whether this function executes speculatively or not: the predicate input to *foo()* may be used to only allow its execution when the predicate is true, thereby providing no speculation. Alternatively, the function may start executing irrespective of the predicate value. In this case, the predicate value will be passed into the subgraph, where side-effect free operations may execute speculatively even before the predicate value becomes available (all side-effecting operations will always be predicated).

Another advantage of having control flow converted in to boolean predicate expressions is the ability to perform *control dependence analysis* to identify regions of code that are control-flow equivalent and may therefore execute in parallel (provided all state and dataflow dependences are satisfied). Consider the *bar()* function in the CDFG (Figure 1a). Despite the aggressive branch prediction, a superscalar processor will not be able to start executing the *bar()* function until it has exited the loop. Similarly, when the *if* branch is predicted-taken, the superscalar processor must switch from executing multiple dynamically unrolled copies of the for-loop and instead focus on executing the control-flow within *foo()*. This is because a conventional processor can only execute along a *single flow of control* [12].

The VSFG on the other hand can use control dependence analysis to identify the control-flow equivalence between the contents of the for-loop and the *bar()* function, and so long as the dataflow and state-ordering dependences are resolved, the *bar()* function may start




**(a)** Performance in Cycle Counts vs LegUp.




**(b)** Energy Consumption comparison vs LegUp and an Altera Nios IIf in-order Processor.

**Figure 4** Performance and Energy Comparison of VSFG, normalized to LegUp results.

executing in parallel with the for-loop. Similarly, the contents of the *foo()* subgraph can also execute in parallel with its parent graph. If we combine this concurrency with loop unrolling, it becomes possible in Figure 3 to execute multiple copies of the loop and *foo()*, in parallel with the execution of *bar()*! This ability to execute multiple regions of code concurrently is equivalent to enabling execution along *multiple flows of control*, and can potentially expose greater ILP than even a superscalar processor.

## 3 Evaluation Methodology & Results

To evaluate our IR and execution model, we implemented a HLS toolchain to compile LLVM IR to custom hardware. We present results for 3 versions of our VSFG hardware: VSFG_0 has no loop unrolling/flattening, VSFG_1 has all loops unrolled once, and VSFG_3 has all loops unrolled thrice. We compare our results against LegUp [4], an established HLS tool that utilizes the CDFG IR and static scheduling (Figure 4). The input LLVM IR to both toolchains was optimized with '-O2' flags, and with no link-time inlining or optimization. All generated circuits were targeted for implementation on an Altera Stratix IV GX FPGA.

We also compare performance against two conventional processors: an Intel Nehalem Core i7 Processor – simulated using Sniper from Intel [5] – as well as an Altera Nios IIf processor, also implemented on the Altera Stratix IV FPGA (Figure 5). Both processors are configured with perfect L1 caches and a hit latency of 1 cycle. Six of the benchmarks used are part of the CHStone HLS benchmark suite [10], while two other are home grown: *bimpa* is a neural network simulator, and *epic* is a matrix transpose function – both of the latter wer selected specifically because they have complex and data-dependent control flow.

Our generated hardware achieves a geometric mean speedup of 1.55× (max 4.05×) over equivalent hardware generated by LegUp (Figure 4a), and is able to better approach (in some cases even exceed) the performance of the Intel Core i7 (Figure 5). While this performance

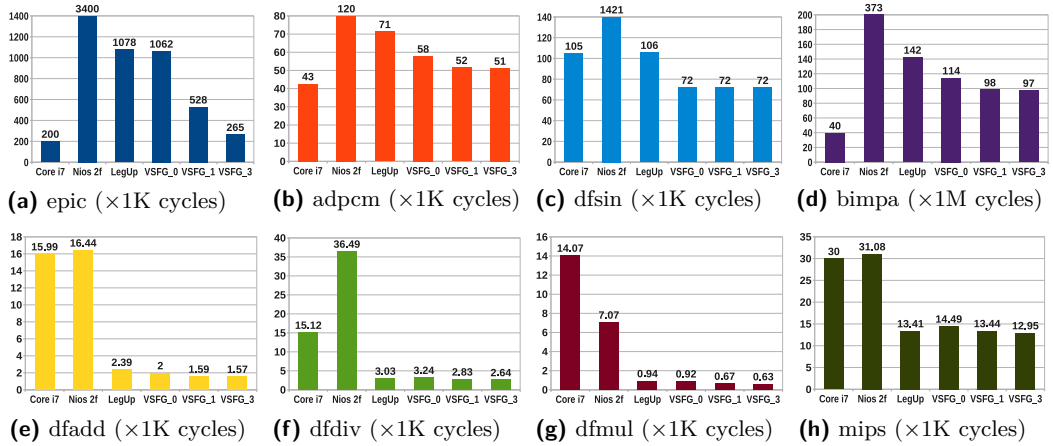



**Figure 5** Performance Comparison (Cycle Count) vs an out-of-order Intel Nehalem Core i7 processor, and an Alteral Nios IIf in-order processor.

incurs an average 3× higher energy cost than LegUp, the VSFG-based hardware's energy dissipation is still only 0.25× that of a highly optimized in-order Nios IIf processor (Figure 4b).

## 4 Conclusion

We combine static-dataflow execution model with a new compiler IR in order to match the sequential performance of superscalar processors in custom hardware. The hierarchical and control-flow agnostic nature of the VSFG not only enables the exploitation of ILP from across multiple levels of a loop nest, but also enables control-dependence analysis and allows execution along multiple flows of control, potentially exploiting more ILP than is theoretically possible for even superscalar processors.

Our results show a performance improvement of as much as 35% over LegUp, at a 3× higher average energy cost. This performance is close to what is achieved by an Intel Nehalem Core i7, while being only 0.25× the energy cost of an in-order Altera Nios IIf processor. We believe that our new IR, coupled with the Spatial Computation execution model is a promising step towards addressing the problem of dark silicon by facilitating the development of high-performance custom hardware.

## References

1 Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

2 M. Budiu, P.V. Artigas, and S.C. Goldstein. Dataflow: A complement to superscalar. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 177 –186, march 2005.

3 Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 14–26, New York, NY, USA, 2004. ACM.

4 Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-

based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

5 Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

6 Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

7 N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, pages 86–95, March 2011.

8 Ed Grochowski and Murali Annavaram. Energy per instruction trends in intel® microprocessors. 2006.

9 Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10114–, Washington, DC, USA, 2004. IEEE Computer Society.

10 Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *ISCAS'08*, pages 1192–1195, 2008.

11 Srikanth Kurra, Neeraj Kumar Singh, and Preeti Ranjan Panda. The impact of loop unrolling on controller delay in high level synthesis. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 391–396, San Jose, CA, USA, 2007. EDA Consortium.

12 Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 46–57, New York, NY, USA, 1992. ACM.

13 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

14 Alan C. Lawrence. Optimizing compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, December 2007.

15 Daniel S. McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 241–252, New York, NY, USA, 2013. ACM.

16 R. Namballa, N. Ranganathan, and A. Ejnioui. Control and data flow graph extraction for high-level synthesis. In *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pages 187–192, 2004.

17 G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M.B. Taylor. Conservation cores: reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News*, 38(1):205–218, 2010.