MEMORY ACCESS PARALLELIZATION IN HIGH-LEVEL LANGUAGE COMPILATION FOR RECONFIGURABLE ADAPTIVE COMPUTERS

Hagen Gädke

Integrated Circuit Design (E.I.S.)
Technische Universität Braunschweig
email: gaedke@eis.cs.tu-bs.de

Florian Stock, Andreas Koch

Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt
email: {stock|koch}@esa.cs.tu-darmstadt.de

ABSTRACT

Control-memory-data flow graphs (CMDFGs) are a unified intermediate representation for compiling high-level languages onto reconfigurable adaptive computing systems. We present both their initial construction as well as transformations for parallel memory accesses. The impact on a number of applications is examined, also considering the effect of caches on acceleration efficiency.

1. INTRODUCTION

A significant research effort has been made in academia and industry to allow the programming of reconfigurable *adaptive computing systems* (ACS) from just a single high-level language description without requiring specialised knowledge from the software developer. One of these efforts is COMRADE [11], a system that has been under steady development since 2001. It aims at efficiently programming an ACS from a C description, without imposing constraints on the language features used or requiring additional source-level annotations from the programmer. C is not treated as a hardware-description language. Instead, the compiler attempts to map the *semantics* of the source program to an efficient combined hardware/software implementation.

In this paper, we focus on a recent enhancement to COMRADE that deals with the *parallel* execution of memory accesses (MA). Our ACS prototypes have long had that capability in hardware [12], however it is just being exploited by the compiler. The method presented here is capable of handling MAs via arbitrary pointers, and thus not restricted to loops with constant bounds and affine index variable updates, as many other approaches. We will also show how the parallelization interacts with the speculative execution and dynamic scheduling supported by our compute model and microarchitecture.

2. RELATED WORK

Efforts to compile C into hardware attack the problem from a variety of angles: Some attempt to treat the input program as a hardware description with C syntax [9], others do attempt to implement C semantics, but only support a subset of the language [17] [16], e.g., supporting only pointers that can be statically analysed at compile time.

Prior work on tools that attempt to compile ANSI standard C without restrictions include GarpCC [4] and Nimble [15]. These tools had practical limitations with regard to the supported C operators and only considered inner loops for acceleration, while COMRADE allows even complex nested structures. Their compute models are also different [13] and they could not efficiently deal with variable-latency operators, a limitation which also extends to cached-memory accesses. A cache miss would result in the entire reconfigurable compute unit (RCU) being stalled. COMRADE uses a significantly more powerful model that supports multiple threads of execution, halting just those actually affected by the current delay.

Of the related approaches, the CASH compiler [1],[2] with its Pegasus intermediate representation (IR) is the one most similar to our work. While there are some obvious differences (CASH targets asynchronous ASICs, not reconfigurable devices), it also relies on token-based dynamic scheduling. However, the model used in COMRADE not only supports the activation of operators, but also the cancellation of misspeculated computations, even across basic block boundaries (important for nested control structures) [7]. MAs play an important role in both systems: CASH explicitly models memory dependencies in its IR and supports speculative writes and run-time disambiguation using a load-store queue. Partial redundancy elimination is used to remove superfluous accesses. Orthogonal to that work, COMRADE uses its IR, the control-memory-data flow-graph (CMDFG), to *parallelize* MAs while still respecting access ordering constraints.

The COMRADE flow itself has been described in greater detail in other work [11]. For the following discussion, it suffices to know that it translates C programs into CMDFGs, which are then optimised and transformed before being exported for hardware realisation.

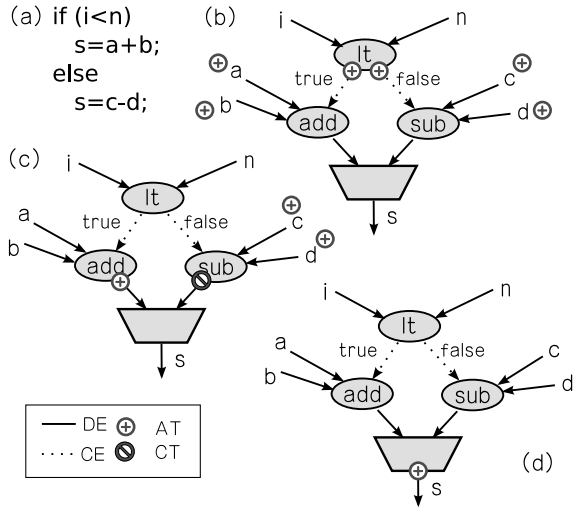

Fig. 1. Speculative execution of alternative branches of an if: (a) source code, (b) CMDFG with initial ATs, flowing downwards, until (c) the false branch gets cancelled. (d) The correct result is passed through the mux.

3. CMDFG DYNAMIC BEHAVIOUR

As will be described in this section, CMDFGs are the *static* compile-time representation of *dynamic* behaviour that will at run-time be realised by the COmrade COntroller MicroArchitecture (COCOMA). In COCOMA, multiple operators may be active simultaneously, leading to a Petri-net like structure with multiple states marked by tokens. However, in addition to these Activate Tokens (ATs), that indicate the presence of a result on an operator output, we also have Cancel Tokens (CTs): If an AT and a CT meet in the same state, they cancel each other out (and lead to the stopping of the associated operator and/or the discarding of its already computed result). In this fashion, a currently misspeculated computation is terminated and the operators are made available for the next incoming data operands. Combinational operators chained into the same clock cycle do not have associated COCOMA states. Instead, their tokens are stored at their fan-in states, as one token for each sequential fan-out of the operator chain. The fan-outs can be activated in parallel. In hardware, each of the token places is realised as a single-bit register, with combinational logic realising the transition and token creation/deletion rules. Control inputs include conditions and operator ready signals (this includes cache hits); control outputs are operator start signals, multiplexer selects and clock-enables for data registers (to update with the correct values). Despite its expressive power (e.g., self-timed dynamic data flow and speculative execution), the hardware area requirements of the controller are quite modest in practice [10].

For brevity and readability, the following description of the run-time mechanisms is stated in prose instead of formal equations. It should suffice, however, to clarify our handling of parallel MAs later in this text. Fig. 1 shows the token flow on a small example: ATs always travel *forward* along data and control edges. Data edges (DE) model true data dependencies. Control edges (CE) run from conditionals (e.g., if, switch, and loop conditions) to the data predecessors of multiplexers. Only if all incoming DEs and CEs of a CMDFG node (except multiplexor nodes) have ATs at their source nodes, is a datum propagated through the hardware operator (indicated by an accompanying AT in the controller). For conditionals, only the true alternative propagates its AT through the multiplexer. The false alternatives generate a CT instead (as soon as the controlling condition has been evaluated, regardless of the availability of all incoming data operands). This CT flows *backwards* along the incoming DEs, cancelling already present data operands for *this* computation of the false branch, or continuing upwards and cancelling the computation of the incoming data operands (if they are not needed elsewhere). Thus, a CT can propagate backwards through an operator only if *all* outgoing DEs of that operator have been cancelled. If the propagation of a CT leads to the cancellation of a conditional, the CT advances *forward* along the CE of that conditional, leading to the cancellation of *all* of the alternatives (this can occur, e.g., in a nested if: once an outer if has been evaluated to true, all computations in the inner if of the else branch can be cancelled [7]). The propagation of a CT stops when it hits an AT, erasing just this one set of computations and freeing the operators for the next incoming data.

4. MEMORY ACCESSES

After establishing the context, we can now discuss our current focus: The integration of MAs with the CMDFG/COCOMA models. In the CMDFG, MAs are also modelled as variable latency operators (due to possible cache misses). Array accesses are transparently split into shift-add address calculations and the MA itself. COMRADE supports speculation across iteration boundaries, overlapping address calculations with other computations (somewhat similar to software pipelining).

On the hardware side, all MAs are handled by MARC [12], a parametrised IP block providing both regular streamed and irregular cached accesses on multiple ports. However, since MARC does not support speculative writes with store-to-load forwarding, all stores are executed non-speculatively (have CEs from their associated conditional in the CMDFG). Loads may selectively be executed speculatively (see below). In addition to data and control dependencies, MAs may also be interdependent. This is generally modelled as read-after-write, write-after-read and write-after-write dependencies [8]. These are reflected as ded-

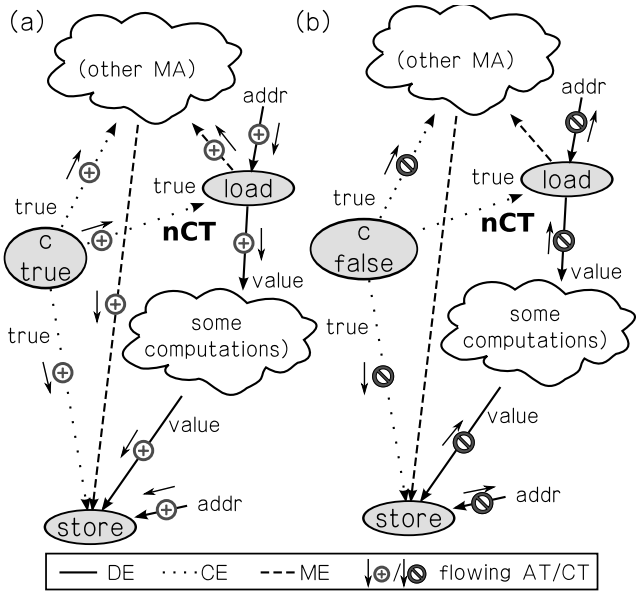

Fig. 2. (a) flow of AT for c == true, (b) flow of CTs for c == false

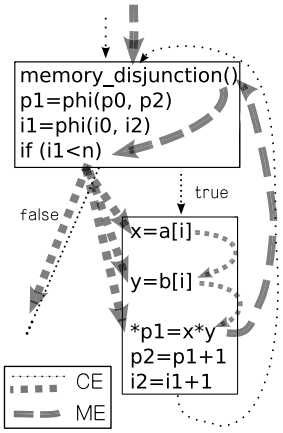

Fig. 3. Adding memory dependencies to MAs in the CMDFG; shown as overlay edges in the CFG

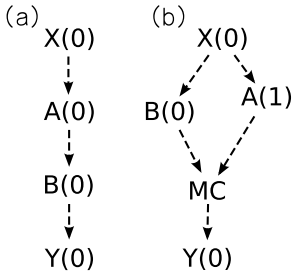

Fig. 4. Parallelizing sequential MAs A and B; numbers indicate associated memory port

icated memory edges (ME) in the CMDFG, imposing an ordering on MAs even in the presence of multiple parallel memory ports. The efficient realisation of these edges in and their interaction with the AT/CT-based control forms the core of this work and will now be discussed in greater detail.

4.1. Memory Sequencing

The serialisation of MAs is realised using the established AT-mechanism. The ATs can now also travel along MEs, sequencing dependent accesses.

Unless speculation of loads is explicitly turned on by the developer, MAs are executed non-speculatively (are the destination of a CE). While we already *could* execute loads speculatively with MARC, doing so would reduce the efficiency of our small cache (see Section 5).

CTs are handled differently for loads and stores (Fig. 2): For the latter, they are processed identically to normal operators, and thus advance backwards along the store's incoming DEs (address and write data). Combined with the backward-moving CTs generated at mux inputs (for untaken alternatives), they cancel load nodes by reaching them via the load's result edge. Note that CTs are generally held at the token place associated with an operator's result. Since stores do not have a result, they are specially endowed with a place to hold a CT incoming over a CE.

Loads are *not* cancelled by the usual method of a CT reaching them along a CE. While this would be desirable (the load could be cancelled earlier, making the operator available early for a new transaction), it would lead to a problem when computing across loop iteration boundaries: when cancelling the load of a prior iteration early, the CTs from the *previous* iteration, still moving upwards from stores or muxes, would have no corresponding datum to erase (the cancelled load never produced one). Thus, CEs ending in a load will not be permitted to carry CTs, indicated by an nCT annotation.

MEs never transmit CTs: They connect memory operations in the same basic block and just sequence them in the required order. The signal whether to execute them at all is indicated by ATs incoming via CEs. If a basic block is *not* executed, contained stores are cancelled using CTs travelling on CEs (for stores) or DEs (for loads), no token traffic occurs on the MEs. Fig. 2 illustrates the flow of ATs/CTs for the true and false cases of a conditional including MAs.

The CMDFG thus allows optimisation passes fine control over the realised hardware: By removing CEs ending at memory operations, the degree of speculation can be increased, while the removal or redirection of MEs between MAs increases the parallelism of those accesses (see Section 4.3).

4.2. Construction of CMDFG Memory Edges

As a baseline for further optimisations, we construct the CMDFG in such a manner that all MAs are correctly serialised in program order (Section 4.3 will introduce parallelism). This initial construction of MEs is done by Algorithm 1; the inserted edges are illustrated in the example in Fig. 3.

Steps 2…9 handle the case of conditionals which contain MAs in only some of their branches: Those branches with MAs use them to correctly forward ATs, but those without break the dependence chain. Thus, we need to add mem-

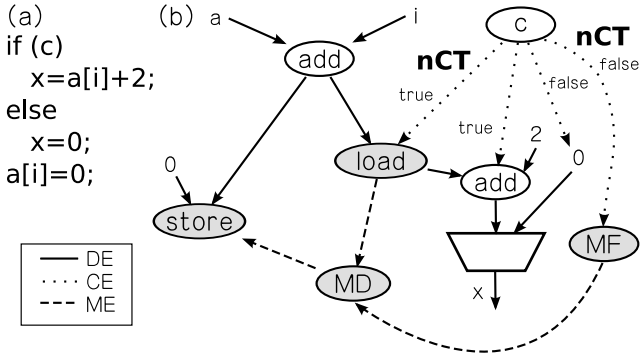

Fig. 5. Inter-block memory flow (a) program code, (b) CMDFG with MF and MD nodes

ory forwarders (MF) to all branches of a conditional *without* a real MA (Fig. 5). The MFs just forward incoming ATs, keeping the dependence chains unbroken even without the presence of actual memory operators. To achieve the AT-only behaviour, we prevent CTs from passing to MFs by qualifying their incoming CE with an nCT annote (Step 6).

Steps 11…19 serialise MAs of different CFG blocks. A memory disjunction (MD) joins the memory flow at CFG join nodes (Step 11…13) analogously to a Φ-node in SSA data flow [5]. It passes a token as soon as one or more of its inputs receive one. Memory statements (MS) can be MAs, MFs, MDs, or MCs (to be introduced in Section 4.3). Thus, Step 15 guarantees that MSs in a successor block are not executed before those of the current block have finished, Steps 16 and 18 connect inter-block memory flow using MEs and MDs.

The rest of the algorithm handles nested loops. As other operators in loops, MAs are also control-dependent [6] on the loop condition. An example of this is shown in Fig. 6.a: The execution of load2 and store3 is controlled by c2, while the data-independent store4 is controlled by the conditional c1.

In a loop nest, we must ensure that MAs in an outer loop start only after the inner loop(s) have *finished*, even if the access itself is only control-dependent on the conditional of the outer loop and data independent of the inner loop. Fig. 6.b shows such a case: store4 can potentially execute before all store3s in the inner loops have completed, resulting in a write-after-write hazard. We resolve this by serialisation, inserting a CE from the conditional of the inner loop to the store in the outer loop (Step 26 in Algorithm 1). For non-memory operators, this would not be necessary, since explicit data dependencies would ensure correct sequencing. However, MAs may have hidden dependencies via memory by accessing the same address (the classic memory disambiguation problem, which is unsolvable for the general case in C). Thus, the controller must conservatively ensure the serial execution of MAs between inner and outer loops.

Algorithm 1 Initial construction of MEs

Require: Each MA has CE to its controlling node
1: **for all** CFG blocks B **do**
2: **for all** Conditionals c at end of B having MA(s) in branch(es) **do**
3: **for all** Branches b of c **do**
4: **if** b has no MA **then**
5: insert MF into b
6: connect controlling branch of MF to MF by nCT-CE
7: **end if**
8: **end for**
9: **end for**
10: Insert MEs between accesses in program order
11: **if** B is a join block **then**
12: create an MD node at its beginning
13: **end if**
14: **if** CFG block B has condition at its end **then**
15: {Bs successor is a join block}
16: Add ME from the block's last MS to the conditional at the end of the block
17: **else**
18: Add ME from the block's last MS to the MD of the successor block
19: **end if**
20: **end for**
21: **for all** loop header blocks $l1$ in CFG **do**
22: **for all** MA m in $l1$ **do**
23: P :=reverse CFG-paths from m's block back to $l1$
24: **if** ∃ a path p in P passing through a sub-loop header **then**
25: $l2$:= first occurring sub-loop header block in p
26: insert nCT-CE from conditional loop exit of $l2$ to m
27: qualify the existing CE from $l1$ to m with nAT
28: **end if**
29: **end for**
30: **end for**

This approach of inserting CEs between MAs in different loop nesting levels scales to arbitrary complexity. However, the handling of ATs and CTs in store4 has just become ambiguous: Which of the two CEs now actually leads to the creation of an AT or CT? Intuitively, the CE from c2 should cause an AT in store4 (since the inner loop is now finished), while an unasserted c1 should send a CT (the outer loop is finished or did not even start). To achieve this, we extend our concept of qualifying CEs, as already introduced in Sec. 4.1: The CE (c2, store4) is qualified with an nCT annote, allowing passage only of ATs (Step 26), while the second CE (c1, store4) is qualified analogously with nAT, propagating only CTs (Step 27). Thus, the ambiguity is resolved.

4.3. Parallelizing Memory Accesses

The CMDFG constructed in this fashion is correct even for complex nested control structures. However, it is overly conservative with regard to the modeled dependencies since it sequences *all* MAs in program order.

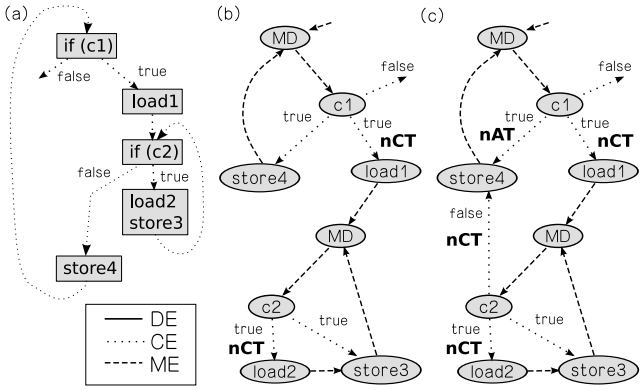

Fig. 6. Nested loops: (a) CFG, (b) CMDFG before Steps 21…30, (c) final CMDFG

This section will discuss how to transform the CMDFG for more parallelism. Note that we will not describe how that parallelism can actually be *discovered* in the program. For that, numerous techniques ranging from simple rules (e.g., consecutive loads and pointers qualified by the restrict keyword are always independent, see Section 5 for their evaluation) to complex analyses (e.g., pointer analysis [20] [19] or loop-iteration space analysis [18] [14]) can be employed. Also orthogonal is the reduction of the number of MAs themselves (e.g., using scalarization [3]).

Algorithm 2 shows our approach. After removing MEs between independent MAs discovered by one of the methods previously, it performs the transformation shown in Fig. 4. However, the parallel accesses now require different semantics of the AT flow: In contrast to the behaviour of our MD node (Sec. 4.2), MAs occurring *after* the parallelized part may now only activate when *all* of the parallel MAs have completed. Thus, we introduce a memory conjunction (MC) node that propagates an AT only if *all* of its predecessors delivered ATs. In addition to this rerouting of the AT flow, the algorithm now also binds accesses to *specific* ports of the memory system. Since MARC already provides multi-port arbitration automatically in hardware, a simple round-robin algorithm suffices for this task. If the number of MAs marked as independent in the CMDFG leads to more parallelism than memory ports are available, the algorithm re-serialises accesses again to keep below that limit. To this end, MEs are inserted to force MAs into program order. Should this lead to an ME fan-in > 1 at an MA, the fan-in edges are combined into one ME by inserting an MC inbetween (waiting for the completion of *all* fan-in MAs before proceeding).

Algorithm 2 Parallelizing Greedy Algorithm

1: **while** exist two independent MAs connected by an ME (A, B) **do**
2: remove edge (A, B)
3: **for all** memory predecessors X of A **do**
4: add ME (X, B)
5: **end for**
6: **for all** memory successors Y of A **do**
7: add ME (A, Y)
8: **end for**
9: **end while**
10: **for all** MAs m **do**
11: **if** m has more than one incoming ME **then**
12: combine these MEs with an MC node
13: assign available memory port numbers to sources of MEs in a round-robin fashion
14: **end if**
15: **end for**
{remove conflicts by serialising}
16: **for all** CFG nodes c **do**
17: **for all** memory ports p **do**
18: **if** p used multiple times in c **then**
19: serialise all MAs using p by inserting MEs between them in program order
20: on ME fan-in > 1 at MA, insert MC between incoming MEs and MA
21: **end if**
22: **end for**
23: **end for**

5. RESULTS

Given an application which performs MAs in $M\%$ of its execution cycles, the maximum speed-up using N memory ports is $\frac{100}{100-M+\frac{M}{N}}$ (Amdahl's Law).

We have examined four kernels [1] and measured the efficiency of using parallel accesses. Table 1 shows both system-level measurements using 4KB on-chip data cache, as well as raw numbers not influenced by cache miss penalties. The cache numbers reflect the MA locality characteristics of the applications. For most cases, cache misses dominate the execution time. The exception is the susan_principle kernel (from MiBench, performing an edge detection on an input grayscale image), which operates here on datasets of 16×16 bytes and thus profits even from our small cache. Applications with irregular access patterns (such as the gfMultiply of pegwit, which performs elliptic curve cryptography) would be better connected directly to main memory using a technique such as FastLane [13].

A second limit for parallelization are the true dependencies in these kernels. Both gfMultiply and susan_principle perform some indirect accesses (the inner MA has to be performed before the outer one). In a similar fashion, gfMultiply has a non-removable WAR memory dependence, also enforcing serialisation. Furthermore, we have parallelized MAs here just *within* CFG blocks. If we would also take control flow into account, susan_principle could be paral-

[1] The final version of the paper will present more benchmarks.

Kernel	Single Port			Parallelized with 2 Ports				
	w/ D$ Miss #Cycles	w/o D$ Miss #Cycles	%Cycles w/ MAs	w/ D$ Miss #Cycles	Disregarding D$ miss penalty			
					#Cycles	Speedup	Theor. max. Speedup	% of max. Speedup achieved
memcopy	366	136	94.1 %	314	72	1.89	1.89	100 %
vec_mult	178	52	92.3 %	143	28	1.86	1.86	100 %
susan_principle	8995	8824	84.1 %	5494	5323	1.66	1.73	90.4 %
gfMultiply	1190	153	44.4 %	1173	136	1.13	1.29	44.8 %

Table 1. Run-time statistics for 1 and 2 memory ports: Execution times (both including and disregarding D$ misses), fraction of execution spent on MAs, achieved speed-up, theoretical speed-up, and efficiency of parallelization

Kernel	Data Path				Controller			Total			
	#Slices	#FFs	#LUTs	#MULT	#Slices	#FFs	#LUTs	#Slices	#FFs	#LUTs	#MULT
memcopy	2139	1985	1176	0	572	281	1077	2617	2270	2067	0
vec_mult	1721	1633	984	12	492	245	929	2109	1885	1701	12
susan_principle	6679	5154	3144	3	1710	998	3204	8448	6152	6262	3
gfMultiply	1523	1793	616	0	473	235	897	1908	2027	1282	0

Table 2. Area required on a Xilinx Virtex II Pro device

lelized to an even greater degree. Despite these limitations, our techniques have achieved a large part of the theoretically achievable speed-up for both the extreme case (memcopy and vec_mult are perfectly parallelizable) as well as for a real-world non-scientific kernel (susan_principle). Chip area data for the kernels is given in Table 2.

6. CONCLUSION AND FUTURE WORK

CMDFGs are an expressive intermediate format for hardware compilation that cannot only model the activation of variable-latency operators, but also their explicit cancellation, making them available for the next set of inputs.

We have shown how to formulate memory dependencies in the CMDFGs and their interaction with control structures such as nested conditionals and loops. As a first application of representing data, control and memory flow in such a unified representation, we proposed parallelizing memory accesses. We demonstrated that we can achieve a large fraction of the theoretically achievable speed-up both for synthetic benchmarks as well as a real application.

Future work will extend to more tightly interweave memory and control flow to allow inter-block parallelization. We also plan a memory system supporting fully speculative accesses (even writes), which will further increase parallelism.

7. REFERENCES

[1] M. Budiu. *Spatial Computation*. Ph.D. thesis, Carnegie Mellon University, Computer Science Department, 2003.

[2] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *CGO*, pages 216–227. 2003.

[3] D. Callahan, S. Carr et al. Improving register allocation for subscripted variables. *SIGPLAN*, 25(6):53–65, 1990.

[4] T. Callahan, J. Hauser et al. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, 2000.

[5] R. Cytron, J. Ferrante et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS*, 13(4):451–490, 1991.

[6] J. Ferrante, K. Ottenstein et al. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[7] H. Gädke and A. Koch. Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In *ARC*. 2008.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2002.

[9] Impulse Accelerated Technologies, Inc. Impulse C - http://www.impulsec.com.

[10] N. Kasprzyk. *COMRADE – Ein Hochsprachen-Compiler für Adaptive Computersysteme*. Ph.D. thesis, Technische Universität Braunschweig (Germany), 2005.

[11] N. Kasprzyk and A. Koch. High-Level-Language Compilation for Reconfigurable Computers. In *ReCoSoC*. 2005.

[12] H. Lange and A. Koch. Memory Access Schemes for Configurable Processors. In *FPL*, pages 615–625. 2000.

[13] H. Lange and A. Koch. An Execution Model for Hardware/Software Compilation and its System-Level Realization. In *FPL*. 2007.

[14] C. Lengauer. Loop Parallelization in the Polytope Model. In *Intl. Conf. on Concurrency Theory*, pages 398–416. 1993.

[15] Y. Li, T. Callahan et al. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*, pages 507–512. 2000.

[16] Mitrionics. Mitrion-C - http://www.mitrionics.com.

[17] E. M. Panainte. *The Molen Compiler for Reconfigurable Architectures*. Ph.D. thesis, Technical University Delft, 2007.

[18] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13. 1991.

[19] M. Shapiro and S. Horowitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, pages 1–14. 1997.

[20] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *POPL*, pages 32–41. 1996.