An Improved Intermediate Representation for Datapath Generation

Nico Kasprzyk, Andreas Koch, Ulrich Golze, Michael Rock

Department of Integrated Circuit Design (E.I.S.)
Technical University od Braunschweig
Mühlenpfordstraße 23, D-38106 Braunschweig
Email: {kasprzyk, koch, golze, rock}@eis.cs.tu-bs.de

Abstract: Traditionally, computer performance is increased by using faster or additional processors. In contrast to this approach, *adaptive computers* accelerate applications by partially executing operations on reconfigurable hardware. To make these machines actually accessible to software programmers, powerful automatic compile flows are required. This paper motivates a new intermediate representation aimed at compiling from C source code to hybrid hardware/software executables for adaptive computers. Control and data flow are handled in a unified manner which simplifies optimizations and also improves the actual generation of hardware data paths. Furthermore, dedicated constructs are available to express speculative execution characteristics.

Keywords: *Adaptive Computers, compiler, intermediate representation, hardware synthesis*

1 Introduction

Beyond conventional microprocessors and DSPs, reconfigurable (sometimes also called *adaptive*) processors offer a chance at even higher computation power. These computers allow the structural adaptation of their architecture to the requirements of the currently executing algorithm. Commonly, a reconfigurable compute unit (*RCU*) is associated with a conventional processor as co-processor or function unit. In this arrangement, the conventional processor executes administrative operations (OS functions, high-level I/O), while the compute-intensive parts of an application (so called *kernels*) are implemented in the reconfigurable part of the system.

One of the reasons that adaptive computers (*ACS*) are still not in widespread use is the difficulty of their programming. In general, this requires experience beyond conventional software programming, namely detailed knowledge in hardware architecture and design.

Considerable effort has thus been expended to overcome these problems and raise the abstraction level of programming an adaptive computer closer to that of a conventional one. Examples include translating traditional languages such as C [4,5,11] into efficient hardware/software solutions. In these flows, the task of a compiler is the partitioning of an application into hardware (*HW*) and software (*SW*) parts and the generation of interfaces between them. Additionally, the actual reconfigurations have to be scheduled.

One of the more crucial issues in such an automatic compile flow is the choice of an appropriate intermediate representation (*IR*) for the program under compilation. Such an IR has to be amenable both for representing as well as for consistently transforming hardware and software blocks.This paper describes an newly developed IR which forms the backbone of our research in hybrid hardware/software compilers.

2 Discussion

Over the years, quite a few IRs have been devised for hardware generation from high-level languages such as C. Often, they were adopted from other research areas, commonly from work dealing with compilation for parallel computers. The requirements in these areas are quite similar to the ones for compilers targeting adaptive computers.

However, some issues merit special attention in an ACS compiler: Since most RCUs are particularly adept at implementing fast data paths, a simple yet powerful handling of data flow in the IR is highly desirable.

HW synthesis projects often employ IRs based on control data-flow graphs (*CDFG*) [6] as well as on control flow graphs (*CFG*) augmented with def-use chains.

While we could also have taken this approach, it would have had some limitations: C programs are not easily represented in CDFGs without additional conversion steps (e.g., the dismantling of do-while loops into conditional branches). Furthermore, nodes for high level constructs such as while loops are entirely missing and complicate later high-level optimizations (e.g., loop restructuring).

Such operations are better performed on a more control-oriented IR such as a CFG. The corresponding data flow can be derived this by an analysis of definitions and usage of variables in individual CFG blocks. Accesses to the same variable are chained together by def-use chains. This approach can also be exploited in optimizations for HW realization, e.g., the reduction of operator and variable bit width [14].

While it appears that such an extended CFG is well suited for our purposes, this turns out to be false when attempting to realize transformations on the IR which also modify the data flow. Such transformations include, e.g., even steps as simple as loop unrolling.

For these operations, data dependencies are modified and the chains between variable definitions and uses have to be updated for all variables affected. Furthermore, the fact that the CFG does not directly represent data dependencies in the IR requires the maintenance of a dedicated data structure in the compiler.

To alleviate these weaknesses, we advocate the use of an IR based on the static single assignment form (*SSA*) [1] for HW generation. Here, the data dependencies are better integrated with the control representation.

To establish a context for this discussion, Section 3 gives an overview of the entire compile flow we developed for the COMRADE system (Compiler for adaptive Systems). Section 4 describes the Data Flow Controlled SSA form (*DFCSSA*), our proposed SSA variant. Some examples show the advantages of DFCSSA when used for hardware generation. Section 5 discusses the overhead incurred by DFCSSA over the traditional SSA form.

3 Compiler architecture

The aim of COMRADE (Figure 1) is the automatic generation of hybrid HW/SW executables for ACSs

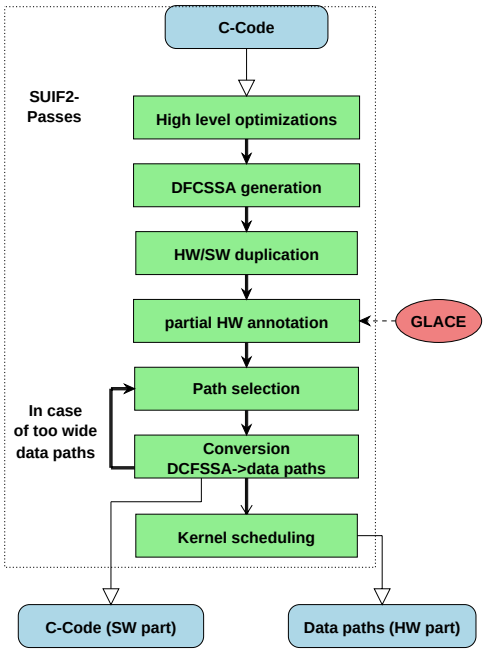

Fig. 1: Compile flow

from traditional non-annotated C source code. It is based on the compiler framework SUIF2 [11] and augments it by adding the DFCSSA form. The actual conversion to DFCSSA form takes place after the C source code has been parsed and been subjected to high level optimizations as described in [1] (e.g., constant folding). The succesive compiler passes then all operate on the DFCSSA representation.

The compiler chooses operations for a later HW implementation at a representation level similar to that of C. Here, high level constructs such as while loops and if branches are still existent in the IR. Now an annotation pass determines various HW characteristics (delay, area, etc.) for all atomic operations (e.g., additions, multiplies etc.). The actual HW/SW partitioning, which occurs at a later phase, relies on the information collected here. The HW characteristics of the individual atomic operations will not be affected by the later transformations (e.g., loop unrolling, dismantling of switch statements to if statements). Hence, this step can proceed at an early compilation stage.

The source for the operator HW characteristics is the module generator system GLACE [12], which also provides estimates in addition to the module netlists. Access to these services is offered by the interface FLAME [10]. In addition to the HW characteristics, the partitioning relies on profiling results for its decisions. At this time, we use a dynamic profiling approach, but

intend to switch to data-independent static profiling [3,17] in the future.

As stated previously, the partitioning concentrates on compute kernels which generally consist of (possibly nested) loops. However, a loop nest is not considered atomically for hardware realization on the RCU. Instead, the partitioning algorithm may decide to only consider inner loops or even only specific execution paths within a loop. The latter capability is very powerful, since it allows the acceleration of loops that contain HW-infeasible operations (e.g., high-level I/O such as printf for an error message). As long as only HW-feasible paths are actually taken at run-time, execution can remain in HW. Of course, once the exceptional condition does occur, the original instructions still have to be executed in SW.

To this end, all candidates (loops and individual paths) for a possible HW execution will be duplicated in the CFG at the path selection compiler stage. This occurs using a technique similar to that used in GarpCC [5]. By inserting new nodes at the fork and join points, the choice between HW and SW execution of the kernel can be made at run-time. Consider the example shown in Figure 3: Two nested loops are duplicated. Nodes 7 and 9 act as fork points and hold the run-time HW/SW mode switches, while nodes 9 and 10 act as join points.

In addition to this run-time switchability, the duplication also allows us to fall back on the original SW version at compile-time if a later compiler stage deems a HW implementation infeasible after all. This might occur, e.g., if an individual kernel does not fit on the RCU anymore when all other candidates have been considered, or if the use of a HW kernel would lead to inefficient reconfiguration scheduling (configuration thrashing).

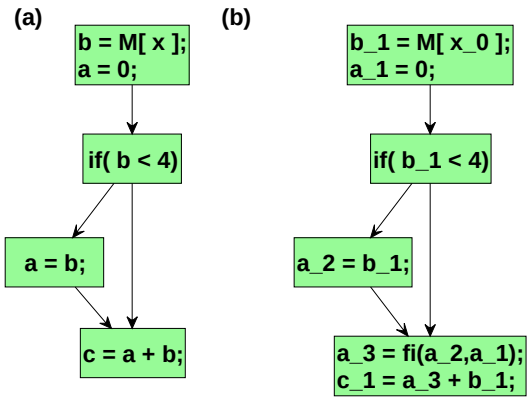

Fig. 2: CFG (a) and CFG in SSA-Form (b)

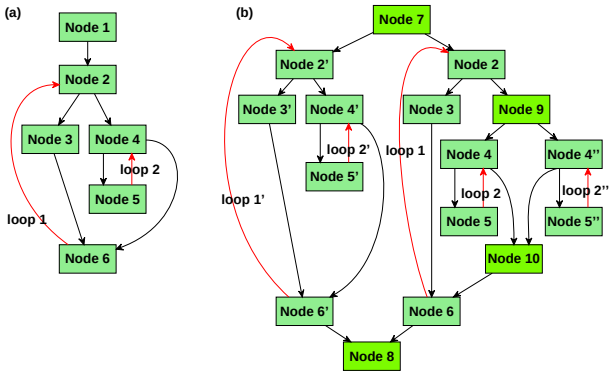

Fig. 3: Loop duplication

Figure 3 also shows that the duplication algorithm not just duplicates the outermost loop, but also all inner loops recursively. Thus, the size of the HW kernel can be grown as appropriate starting from the innermost duplicated loop and working outwards.

Appropriate interfaces will be added on both the HW and SW sides at the fork and join nodes. E.g., the SW side of a SW-to-HW fork node will hold code to transfer live SW variables to HW registers. At the HW side, the same information is used to actually generate these CPU writable registers.

The final compiler pass translates the CFG from its DFCSSA form into an actual HW data path for the target RCU. These data paths are then scheduled and processed by lower-level tools [9].

4 Representation in SSA form

The hybrid HW/SW compiler we propose relies heavily on a suitable IR: The IR must be able to easily represent the source language at multiple abstraction levels as well as efficiently describe data and control flow even when subjected to complex transformations. As already stated previously, the latter demand makes the def-use-chain representation undesirable. Before continuing to explain our DFCSSA variant, we will now introduce some of the basics of the original SSA form.

When representing a program in SSA form, every write access to a variable leads to the creation of a new name for the destination variable. E.g., in Figure 2, the assignments to variable a become definitions of a_1 and a_2. At nodes where multiple control flows join in the CFG, it must be determined which of the multiple definitions to use when a variable is read. To this end, so-called Fi statements are inserted to merge the appro-

```
for( i=0; i< 10; i++ ) {
  a[i] = 0;
}
for( i=0; i<10; i++ ) {
  b[i] = 0;
}
```

Fig. 4: Example program 1

priate variable definitions. In the example, the definition of a_3 is created by a Fi statement which resolves the value from the definitions a_2 and a_1.

Even in this basic state, the SSA form is advantageous when aiming for HW implementation of the CFGs. One effect is the shortening of individual variable life times, which can lead to the removal of data dependencies. The example code in Figure 4 shows the usage of variable i in two independent loops. After the conversion into SSA form, a later compiler pass can easily recognize that these two loops can be executed independently. Since no other data dependencies exist, the two loops could also execute in parallel.

5 DFCSSA form

For COMRADE, we extended the original SSA. The first change affects the location of the Fi statements in the CFG nodes. In the original SSA form, Fi statements to resolve variable definitions were placed in all nodes where control flow edges join. In DFC-SSA, however, we place Fi statements only in join nodes where the variable values are actually read, and even then only for the *specific* variables read. Consider the following example: In the original SSA form, the sample program shown in Figure 5 would result in the IR depicted in Figure 7. Fi statements are inserted at all joining edges, even though some resolved variables (a_4) are not used anymore in the program. This is

```
i = 0;
a = 0;
do {
  if(i<15) {
    if(i<10) a = 1;
  } else     a = 2;
  c = a+1;
  i=i+1;
} while( i < 100 );
```

Fig. 5: Example program 2

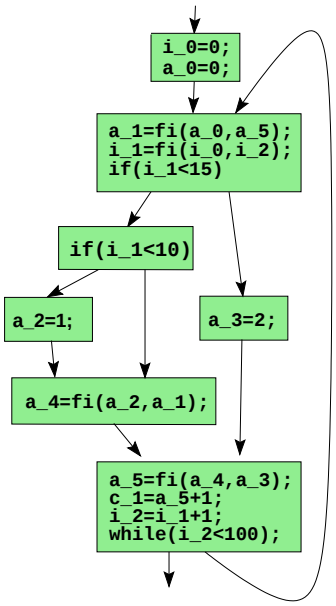

Fig. 7: Common SSA representation

avoided in the DFCSSA form, where redundant Fi statements do not occur anymore (Figure 6).

The insertion of Fi statements right at the point where a variable is used can temporarily increase the number of Fi statements over that in the common SSA form. This behavior is also shown in Figure 6 by several uses of the variable i. The Fi statements for the definitions of i_1 and i_3 have the same arguments. However, this seemingly wasteful structure actually has advantages (ability to backtrack similar to a use-def-chain) for later optimization steps. Furthermore, it

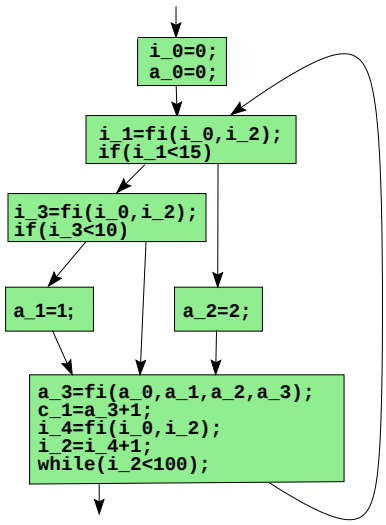

Fig. 6: DFCSSA representation

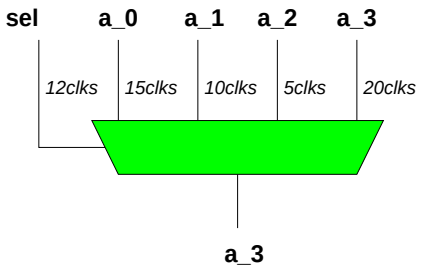

Fig. 8: Realization of fi statements as multiplexer

does not affect the size of the generated HW, since Fi statements with the same or supersets of existing argument lists will be merged into a single Fi statement in the creation of a data flow graph (*DFG*) from the CFG in DFCSSA form. These effects are quantitatively examined in Section 7.

5.1 Partial processing

As described previously, only those parts of the CFG actually feasible for HW implementation will be processed further in the custom flow. The SW part will be extracted and handed to a standard SW C compiler for translation to object files.

With this intent, the DFCSSA translation can be concentrated only on those parts of the CFG that will end up in HW. Thus, COMRADE is working with a partial DFCSSA representation which keeps the SW regions of the CFG untouched. The same idea also applies to other transformations (e.g., matching operator and variable bit widths to the specific data at hand), which are also limited to the HW parts.

6 Translation to a DFG

Since the DFCSSA form implicitly represents the data dependencies of a program, the actual creation of a data flow graph is simplified.

Initially, Fi statements with identical argument lists will be merged (Section 6.1). Then, the translation pass connects the definition of each variable with all of its uses (reads). The resulting DFG now contains all operation of the original CFG. Only Fi statements, which are artifacts from the translation into DFCSSA form, have to be handled separately. Mirroring their purpose of selecting one value from a number of alternatives, they are realized as multiplexers. Thus, the Fi statement for a_3 in Figure 6 is mapped to a multiplexer as shown in Figure 8.

Such a realization allows the exploitation of fine-grained parallelism in that all of the data input values into the multiplexers can be calculated in parallel. This also has a speculative character, since, as soon as the control input sel has stabilized, the calculations on the unselected inputs can be aborted.

Using Figure 8 as an example, if the value for sel is valid after 12 clock cycles and selects a_0 as output for the multiplexer, the final multiplexer output value would be available at a time of 15 cycles. Other, possibly longer running calculations (such as that of a_3) can be safely aborted.

6.1 Efficient multiplexer synthesis

One property of the DFCSSA form is the resolution of a variable value from multiple definitions right before a read of the variable. In HW terms, this approach can reduce the multiplexer delay and also the compilation time itself because an explicit multiplexer optimization is no longer required.

As an example, if the CFG shown in Figure 7 would be naively translated into a DFG, a cascade of three multiplexers (Figure 9a) would result. Obviously, a faster realization using a balanced tree is possible (Figure 9b). However, starting with the original SSA form, this faster HW could only be achieved after running a dedicated multiplexer balancing compiler pass.

In contrast, the DFCSSA form as in Figure 6 creates a a single Fi statement only when required (due to a read). Since at that point the precise number of inputs to the multiplexer is known (four in the example), a perfectly matching HW instance can be generated atomically using the module generators. Furthermore, since they might be able to exploit architectural features of the target RCU aimed at efficiently implementing wider multiplexers, the resulting HW could be even faster and smaller than the balanced tree of 2-input multiplexers shown in Figure 9b.

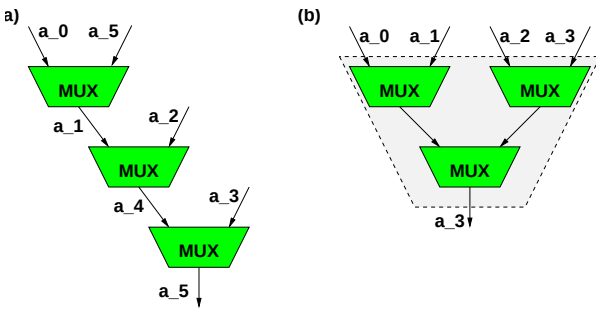

Fig. 9: Multiplexer implementation

Application	#statements w/o SSA	SSA-Form #fi statements	DFCSSA #fi statements	DFCSSA #fi statements, optimized	Increase unoptimized	Increase optimized
adpcm	171	12	10	7	-16%	-41%
pegwit	2154	163	221	135	26%	-17%
jpeg	5027	469	697	441	48%	-6%
fft	104	16	19	11	19%	-31%
pgp	12520	1284	1574	1051	22%	-18%
bytemark	539	32	42	25	31%	-22%
wavelet	312	30	51	25	70%	-17%

Fig. 11: Overhead for DFCSSA form

6.2 Parameter transfer from SW to HW

Often, one input of the multiplexers resulting from the Fi statement is a value transfer from a SW variable to a HW register. Assuming that the program code in Figure 5 (but without the initialization of i and a) is intended for HW execution, the multiplexer for a_3 would have one external input from the SW part, and four HW-internal inputs (Figure 10). A buffer register accepting the parameter from SW would be used only once in this example. namely for the value transfer at the beginning of the HW execution.

This can be optimized, however: In our HW architecture, most operators are being followed by a register for pipelining purposes. In these cases the original buffer input register can be removed and the parameter value is directly written into the real hardware register that will be used during the computation.

6.3 Input selection

The sel input of the multiplexers is primarily controlled by a datapath-external FSM. The construction of that FSM is also amenable to various optimizations. E.g., while it would be possible to simply model the FSM in a fashion closely following the original program flow,

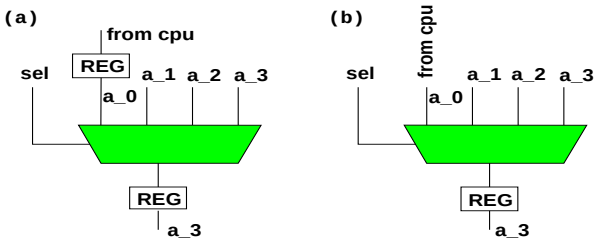

Fig. 10: Multiplexer optimization

this would complicate exploitation of the speculative execution technique described in Section 6.

An alternative approach is used in the Pegasus IR [4], which also uses a specialized variant of the SSA form. There, the multiplexers' select inputs are generated using expressions associated with the corresponding Fi statements: If that expression evaluates to True, the first data input of a 2-input Fi statement is selected, otherwise the second data input is propagated. This approach is primary useful for Fi statements with two inputs. In Pegasus, these multiplexers (and their selecting expressions) are merged later, requiring an extra data flow analysis step.

Furthermore, the simple representation used by Pegasus is unable to accommodate the *N*-input Fi statements (*N*>2) which are the usual case in a DFCSSA representation. As a solution, we abandon the node-local information in favor of a kernel-wide data structure from which arbitrarily complex conditions can be extracted. This data structure is based on a control dependence graph (*CDG*) [15] and covers the conditionals of all Fi statements in the CFG. Due to its global nature, inter-statement optimization and analysis passes are also simplified.

7 Overhead incurred in DFCSSA

The price the DFCSSA form pays for the advantages described in the previous sections is an increase in the number of Fi statements. However, we will now demonstrate that this does not affect the quality of the generated hardware due to the way the DFG is created.

For a set of sample applications, Figure 11 lists the numbers of Fi statements both for the SSA and DFCSSA forms. The total number of statements of the

entire application before transformation to any SSA form is also shown. For counting purposes, a statement is any C assignment, loop, branch, and label. As can be seen, the DFCSSA form initially requires a larger number of Fi statements that the original SSA form. However, after optimizing Fi statements with identical arguments and reusing previously merged Fi statements as subexpressions (in the case of argument supersets), the total number of Fi statements decreases considerably (even below those of the original SSA form). Thus, the overhead of DFCSSA over SSA materializes in a temporary increase in memory consumption.

8 Conclusion

With DFCSSA, we have introduced a new variant of the classical SSA form which we find very well suited for hybrid hardware/software compilation tasks. It allows a more concise representation of the program structure as well as fast optimizations with low administrative overhead. The main disadvantage compared to SSA, the increased compile-time memory requirements, is only temporary and no longer applies after preparing the program representation for hardware generation.

References

[1] Appel, A., Modern Compiler Implementation in C, Cambridge University Press, 1998

[2] Bacon, D. F., Graham, S. L., Sharp O. J., Compiler Transformations for High-Performance Computing, ACM Computing Surveys 26(4), 1994

[3] Ball, T., Larus, J., Branch Prediction for free, Proc. of the Conf. on Prog. Language Design and Implementation, 1993

[4] Budiu, M., Goldstein, S., Pegasus: An Efficient Intermediate Representation, Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002

[5] Callahan, T., Hauser, R., Wawrzynek, J., The GARP Architecture and C Compiler, IEEE Computer 33(4), 62-69, April 2000

[6] Drinic, M., Kirovski, D., Behavioral synthesis via engineering change, Proceedings of the 39th DAC, 2002

[7] Gokhale, M.B., Stone, J.M., NAPA C: Compiling for a Hybrid RISC/FPGA Architecture, Proc. IEEE Symp. on FPGAs for Custom Computing Machines, 1998.

[8] Harr, R., The Nimble Compiler Environment for Agile Hardware“, Proc. ACS PI Meeting, Napa Valley (CA) 1998

[9] Kasprzyk, N., Koch, A., Advances in Compiler Construction for Adaptive Computers, The 7th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA, June 26-29, 2000

[10] Koch, A., Kasprzyk, N., Module Generator-based Compilation for Adaptive Computing Systems, IEEE Intl. Symp. on FCCMs, Napa Valley (CA, USA), 2002

[11] Monika Lam, An Overview of the SUIF2 System, ACM SIGPLAN ’99 Conference on Programming Language Design and Implementation, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/tutorial99.ps

[12] Neumann, T., Koch, A Generic Library for Adaptive Computing Environments, Workshop on Field-Programmable Logic and Applications, Belfast, 2001

[13] Li, Y.B., Harr, R., et al. Hardware-Software Co-Design of Embedded Reconfigurable Architectures, Proc. Design Automation Conference, 2000

[14] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., Sherwood, T., Bitwidth Cognizant Architecture Synthesis of Costum hardware Accelerators, IEEE TRANSACTIONS OF COMPUTER-AIDED DESIGN OF INTEGRATED CIRDUITS AND SYSTEMS, VOL. 20, NO. 11, 2001

[15] Muchnik, Advanced Compiler Design & Implementation, Morgan Kaufmann Publishers, 1997

[16] Rock, M., Implementierung einer SSA-Form zur effizienten Erzeugung von Datenflussgraphen für Adaptive Rechner, Diplomarbeit, TU Braunschweig, 2002

[17] Wu, Y., Larus, J. R., Static Branch Frequency and Program Profile Analysis, In 27th IEEE/ACM Symposium on Microarchitecture (MICRO-27), 1994