# Technology in Construction of Compiler for Adaptive Computers

Chandra Srinivas Potluri, Sathish Kumar Konga, Sreedevi Kadiyala

**Abstract:** This paper presents a compiler system for adaptive computing. Our approach Increases the flexibility and usability in a way that allows to port the system to different targets with a minimal effort. Built on an existing design flow, we try to reach a new level of functionality by analyzing and partitioning C programs at the highest possible description level. We show that the analysis on this level is more efficient than on lower ones due to the exploitability of more expressive programming constructs. The improved analysis results combined with a new SSA based algorithm for data path creation can lead to a higher solution quality of the final system configuration.

**Keywords:** Adaptive Systems, Compiler Systems, Control Flow Graph, Data Flow Graphs, Hardware/Software Partitioning, Reconfiguration Scheduling, Static Single Assignment.

## 1. INTRODUCTION

Traditionally, arithmetic performance of computing systems is increased by faster or more processors. Adaptive systems, on the other hand, accelerate programs by executing parts of the algorithm on adaptive hardware. These elements can be dynamically reconfigured during the program run. Some existing research projects in adaptive systems have already demonstrated the advantages: Coupling a MIPS II processor with a special FPGA [BaGS94] in the project BRASS of the University of Berkeley [Wawr00] has resulted in speed-ups of 2 to 10 (simulated). An example of a real system is the Nimble project at Synopsys [Harr98] which was performed in cooperation with the University of Berkeley and our department [Koch96].

## 2. NIMBLE DESIGN FLOW

One of the aims of Nimble is to generate executable software and hardware as quickly as pure software compilation, and not as slowly as the hardware synthesis common today. Such HW/SW programs should be compiled in a time frame of ten minutes instead of several hours. Figure 1 shows the design flow of Nimble. The core compiler, which is the focus of this paper, reads a program described in a high-level programming language. The compiler then analyses the program, partitions it into hardware and software, and generates data paths for the reconfigurable hardware (RL). In parallel, the software part is instrumented with functions for configuring and exchanging with the RL. This extended software part is finally output as C-code.

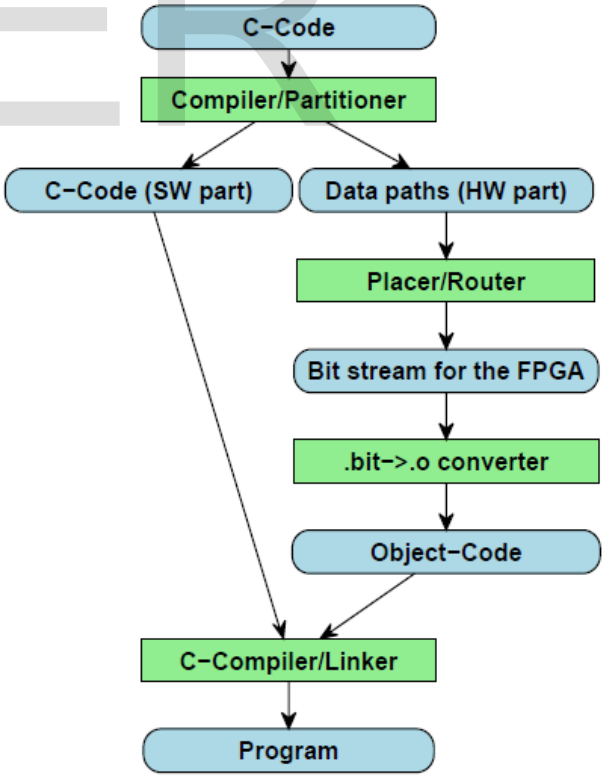


Figure 1: HW/SW design procedure in Nimble

The data paths are pre-placed by a dedicated tool developed specifically for this purpose. Routing is

- *Chandra Srinivas Potluri is currently pursuing Ph.D in Computer Science, from University of Allahabad, India. Working as an Asst. Professor in Debre Berhan University, Ethiopia. E-mail: pcsvas@gmail.com*
- *Sathish Kumar Konga is currently working as an Asst. Prof in Sri Gayatri Degree & PG College, Mulug Road, Hanamkonda, Warangal, Telangana, India. E-mail: konga.knga@gmail.com*
- *Sreedevi Kadiyala Ph.D in Computer Science in University of Allahabad, India. Working as an Associate Professor in Wolkite University, Ethiopia. E-mail: sreedevikadiyala@gmail.com*

performed by the standard Xilinx M3 suite. Afterwards, the bit streams for the RL are compressed and converted into linkable object files. The final program is the result of linking the SW C-code and the HW object files.

## 3. COMPILER ARCHITECTURE

The current compiler [BaGS94] partitions the application into software and hardware and produces Data Flow Graphs (DFG) for the latter. Although it works satisfactorily and obtains good results, it has potential for improvements. Particularly, the heuristics for the selection of program partitions for hardware, as well as the generation of data paths can profit from further work.

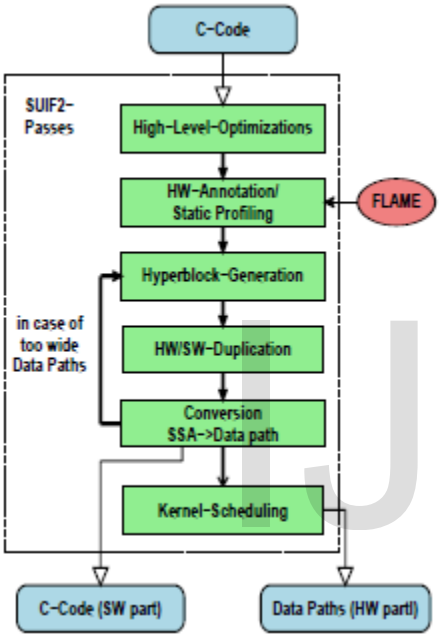


Figure 2: The Compiler C_fas

One weakness is that the characteristics of target RL were not considered in sufficient detail. As an example, the sizes of the hardware components were determined only by looking at SW operations, not the HW functions actually available. E.g., if the library for one type of RL contains a "left shift" cell which might be unavailable on another RL architecture then the compiler would not be able to handle this. Additionally, certain analysis phases rely on out dated methods. This paper discusses multiple routes of approach to expunge these misfeatures and improve the system as a whole.

### 3.1 SUIF2 COMPILER SYSTEM

For this work, we use the SUIF2 compiler system SUIF2 of the University of Stanford [LamM99]. The intermediate representation of program code used by SUIF2 is quite clear and easily expandable. Its modular concept allows quick modifications that are only local in scope. This differentiates it from other well-known compilers such as GCC [GCC00], whose intermediate representation was more optimized for performance than for clarity.

We have also evaluated the SGI-Pro64-Compiler [SGI00] because it supports important structures for us (hyperblocks, SSA form of control flow graphs). But the appropriate methods are unfortunately supplied only on a very low-level intermediate representation that is very close to the executable program. In contrast to [BaGS94], we want to perform optimizations and partitioning on a very high level of program representation. In this approach, we differ from the current Nimble.

### 3.2 Work on High-Level Representations

A high-level representation can express information, which becomes lost or distorted in lower forms. In these cases, it has to be tediously reconstructed by complicated analysis algorithms.

```
switch( anzahl ) {
  case 1: a=1; break;
  case 2: b=1; break;
  case 3: c=1; break;
}
```

```
if (anzahl==1) goto tmp_label1;
if (anzahl==2) goto tmp_label2;
if (anzahl==3) goto tmp_label3;
goto tmp_label4;
tmp_label1: a = (1); goto tmp_label5;
tmp_label2: b = (1); goto tmp_label5;
tmp_label3: c = (1); goto tmp_label5;
tmp_label4:;
tmp_label5:;
```

Figure 3 : Different Representations

The example in Figure 3 demonstrates this: All three cases of the SWITCH statement can be speculatively executed in parallel since the cases are data independent and do not have to be processed sequentially (BREAK). In a lower-level machine-oriented representation (shown at the above of Figure 3), the same information becomes available only after additional analysis steps are executed.

### 3.3 Optimization Steps

Our first optimization steps rely on proven machine-independent methods. We employ Scalarization [CaMT94], Software Pipelining [Much97], Code Movement and other techniques described in [BaGS94]. In this manner, we reduce memory accesses in loops and attempt to create inside loops having a high degree of instruction level parallelism.

### 3.4 Selection of Hardware Building Blocks

Next, the compiler selects operations for a later hardware implementation on the high level representation. This is feasible since HW operations will not be affected by lower level transformations. For evaluating the HW-suitability of SW operations, we rely on estimation data available through the FLAME interface [Koch00]. This flexible interface allows access to hardware-specific libraries, and offers hardware-relevant data such as area and time requirements of operations, availability of synthesizable components as well as chip resources. Partitioning relies mainly on the data collected in this step.

After annotating the operations with this HW information, the program representation is now extended to a control flow graph (CFG). A disadvantage of a CFG on such a high level is that we must consider more types of operations as nodes of the CFG. Thus, additional control structures exist beyond simple branches (e.g., SWITCH, FOR, WHILE statements). However, the advantages described in section 3.2 outweigh this. Furthermore, the CFG needs to be generated only once and adapts to later modifications automatically. The generation of such a CFG is supported by available modules in SUIF2.

### 3.5 Profiling

A static profiling pass follows the annotation of the operations. Here, paths of the CFG are marked with their execution frequency. Note that we consciously decided against a dynamic profiling approach.

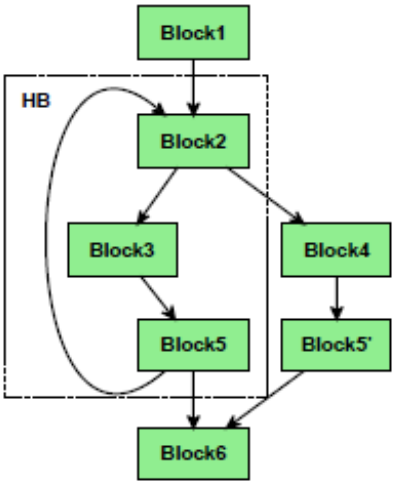


Figure 4: Hyper block with tail duplication

The papers [BaLa93] and [WuLa94] show that static profiling is sufficient for the determination of the most frequented program regions: in the average, 80% of the 20% most-used blocks were determined, in programs from the SPEC benchmarks and various UNIX commands. We can improve this already satisfying result in our hyperblock selection by the fact that we incrementally optimize the appropriate threshold value (both defined later).

### 3.6 Hyperblocks

Now that all necessary information for a further processing is available, we proceed as described in [BaGS94]. We pick regions from the CFG which are especially suited for HW execution. In our case, we concentrate on loops. Before we execute further transformations, these regions are duplicated. If they later turn out to be unsuited for execution in hardware, we can go back to the original untransformed version.

Furthermore, we can use the software version of a loop if we must interrupt the hardware execution, e.g., when the execution hits a HW-infeasible statement in a HW loop. In this case, the SW version performs the corresponding iterations. Specific reasons for exits from a HW-loop include the call of functions which are not or only very inefficiently realizable in hardware. This generally also applies to floating point operations, for which the processor usually possesses a dedicated FPU, or I/O instructions which manipulate HW inaccessible to the RL.

In order to select paths in the loops as regions for hardware execution, they must be evaluated in terms of their "quality". This quality value is proportional to the execution frequency of a path and anti-proportional to the HW-area required. All paths with a quality value above a threshold are combined into a HW block.

If such a block has multiple entry points, we must switch several times between hardware and software execution. These switches require time for data exchange between processor and RL. To avoid this, we rely on the theory of hyperblocks [Mahl96]. These are blocks of the CFG with only one entry point but several exits.

In Figure 4 one recognizes that Block5 was copied to Block5' outside of the hyperblock HB. Without this so-called tail duplication, the hyperblock HB would possess several inputs (Block1 -> Block2, Block4 -> Block5).

If there are several nested loops, which could be selected as hyperblocks exist, this represents a special challenge for our algorithm. In that case, the loops are duplicated in order from outer to inner loops. Contrary to [BaGS94], we do not consider a contained loop as an independent block. Instead, it is considered as normal control flow and evaluated as described before with the hyperblock selection algorithm. Thus, we can select from a larger quantity of potential HW-data paths.

### 3.7 Static Single Assignment Form

After the selection of hyper blocks for HW execution, these program parts are transformed in a way that leads to easily implementable blocks for the target RL. The representation as a DFG is particularly suitable for our algorithm. An important prerequisite is the detailed analysis of the data dependencies of the different variables in the selected blocks. We use a relatively new development from the research in compiler construction in contrast to [BaGS94]. Instead of definition-use-chains of variables used previously, we convert the CFG for the selected blocks into the static single assignment form (SSA) [Much97]. When a CFG is represented by the SSA form, there is only one definition and one use for each variable. Thus, we can considerably simplify different optimizations on this part of the CFG. Sample optimizations using the SSA form are described in [Appe98]. Additionally, dependencies between arguments are resolved. Consider the following program fragment

```
for(i=1; i<100; i++) a[i] = 0;
for(i=1; i<100; i++) b[i] = 0;
```

There is no reason to actually use the same index variable in both loops. In the SSA form, this dependency does not exist and both loops may be executed in parallel. In particular, we use the array SSA form from [KnSa98], which also handles dependencies between different array items. The resulting hardware data paths do not access memory more frequently than absolutely necessary.

From the CFG in SSA form, the DFGs can be easily produced. This is a further advantage especially for our application: Each definition of a variable represents a node of the DFG. An edge of this graph goes from the definition of a variable to its use. DFGs without multiplexers are built from CFGs without branches. We have to insert multiplexers in the DFG and also into the resulting HW data path if joining branches exist in the CFG. These locations of multiplexers are indicated by the location of $\phi$ - functions in the nodes of the SSA form. The decision conditions of these multiplexers are easily derivable from the dominance structure of the CFG [Much97].

### 3.8 Data Path Scheduling

When we have computed all data paths, we can now decide whether each data path actually fits on the target hardware. Otherwise, we must repeat the generation of hyperblocks with a smaller threshold value (section 3.4). Finally, if all data paths fit individually on the target hardware, we can decide further whether it is worth to pack several data paths on the RL at the same time. Helpful for this task is a data path load graph (Figure 5) which indicates the use of data paths as a function of the control flow. The figure shows a possible partitioning of the data path load graph. In the example shown, the entire loop from data_path4 to data_path6 fits at one time on the RL. Thus, unnecessary reconfiguration times are avoided.

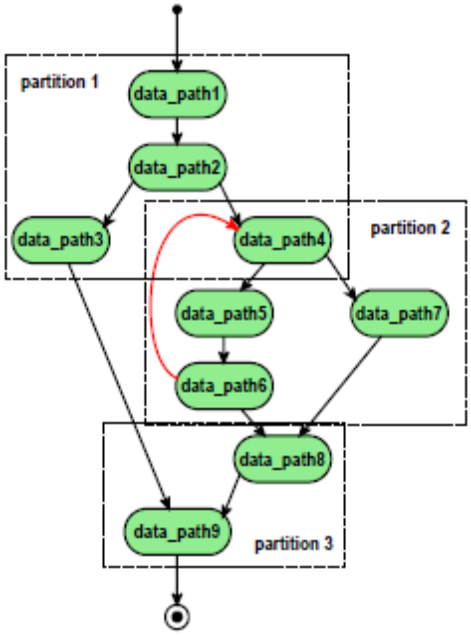


Figure 5: Data path load graph

The algorithm for partitioning the data path load graph must also consider whether it is really worthwhile to implement all possible data paths in hardware. E.g., if the loop specified above would not fit perfectly on the RL and additionally had a high execution frequency, then it could be advantageous to implement only two of the data paths in hardware and one purely in software. Thus, one would avoid "thrashing" between the two RL configurations. Previous work shows that numerous improvements are possible particularly in this area. This will be emphasized in our of further research.

## 4. CONCLUSION

We described an improved approach for a compiler which partitions a high-level language program automatically for partial HW execution on adaptive systems and generates suitable hardware data paths from CFGs. On the basis of existing solutions, we introduced new techniques for a more balanced partitioning and improving data path quality. Further research will actually quantify the advantages in relation to the current system.

## 5. REFERENCES

[Appe98] Appel, A., Modern Compiler Implementation in C, Cambridge University Press, 1998

[BaGS94] Bacon, D. F., Graham, S. L., Sharp O. J., Compiler Transformations for High-Performance Computing, ACM Computing Surveys 26(4), 1994

[BaLa93] Ball, T., Larus, J., Branch Prediction for free, Proc. of the Conf. on Prog. Language Design and Implementation, 1993

[CaHW00] Callahan, T., Hauser, R., Wawrzynek, J., The GARP Architecture and CCompiler, IEEE Computer 33(4), 62-69, April 2000

[CaMT94] Carr, S., McKinley, K. S., Tseng, C., Compiler Optimizations for Improving Data Locality, In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1994

[GCC00] GCC Homepage, http://www.gnu.org/software/gcc/gcc.html

[Harr98] Harr, R., The Nimble Compiler Environment for Agile Hardware", Proc. ACS PI Meeting, http://www.dyncorp-is.com/darpa/meeting/acs98apr/Synopsys\%20for\%20WWW.ppt, Napa Valley (CA) 1998

[HaWa97] Hauser, J. R. and Wawrzynek, J., GARP A MIPS processor with a reconfigurable coprocessor, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa, CA, April 1997

[KnSa98] Knobe, K., Sarkar, V., Array SSA Form and its use in Parallelization, POPL, San Diego 1998

[Koch96] Structured Design Implementation – A Strategy for Implementing Regular Datapaths on FPGAs. In International Symposium on Field Programmable Gate Arrays, Monterey, CA., Feb. 1996

[Koch00] Koch, A., FLAME: A flexible API for Module-based Environments - Users Guide and Manual, http://www.icsi.berkeley.edu/~akoch/research.html#FLAME, Berkeley (CA), 2000

[LamM99] 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

[Mahl96] Mahlke, S., Exploiting instruction level parallelism in the presence of conditional branches, PhD Thesis, University of Illinois at Urbana-Champaign, 1996

[Much97] Muchnik, S. S., Advanced Compiler design implementation, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997

[SGI00] SGI Pro64, http://oss.sgi.com/projects/Pro64/

[Wawr00] J. Wawrzynek, The BRASS Research Project, http://brass.cs.berkeley.edu/

[WuLa94] Wu, Y., Larus, J. R., Static Branch Frequency and Program Profile Analysis, In 27th IEEE/ACM Symposium on Microarchitecture (MICRO-27), 1994