Graph-Based Code Selection Techniques for Embedded Processors
Rainer Leupers and Steven Bashford

INTRODUCTION
Code generation for embedded processors is an important issue. Mainly, due to the memory limitations and performance requirements in these systems. Traditionally methods like tree parsing is not sufficient enough in order to explore all the features. The processors of today often include specialized hardware for certain applications and there is also restrictions on the register use. This together with the requirement on efficient code leads to assembly programming or the need of compiler intrinsics. The portability is effected negatively if you have to program everything or parts in assembly code. The goal is of course to program everything in C or any other high level language. Then you just have to compile it towards different architectures. To make things even worse, many processors also contain instruction level parallelism, i.e. more than one machine operation can be executed in the same clock cycle. The paper from Leupers and Bashford describes two techniques to compile towards specialized architectures, namely SIMD and irregular data paths. The first one is common in media application where the same instruction is applied to many input data. The second one is common in DSP architectures of today, where the registers are more connected to certain functional units.

CODE GENERATION FOR MEDIA PROCESSORS
The processors of today usually have a native word length of 32 bits. This includes that the corresponding register file is also contains registers of the length 32 bit. However, for some media applications, like video and audio, the required precision is only 16 or 8 bit. This leads to a waste in the usage of the registers. A better solution would have been to utilize every 32-bit register as two 16-bit or as four 8-bit registers. This is possible without modifications for the so called trivial SIMD instructions. Example of these are the logic operations AND, OR, NOT and XOR. For instructions like ADD and SUB, things are getting more complicated. We have to get rid of the carry propagation between different words within the same register. The solution is to include SIMD instructions, which perform for example 16-bit addition on the upper and the lower part separately. The problem for the compilers is to explore this feature at compile time. This also put restrictions on the memory alignment. In order to explore the feature of SIMD instructions, it must be possibly to do memory load directly to both the upper and the lower part of the register.

To be able to explore the features of SIMD instructions we have to look at the data-flow-graph (DFG) instead of the data-flow-tree (DFT) as the basic block. The difference between a DFG node and a DFT node is that the fan out of a DFG can be greater than one. I.e. the output from one node, lets say a addition, can just feed one other node as input in the DFT case. In the DFG case, this output can be input to for example two different addition or to a addition, a subtraction and a multiplication node. Traditionally, you transform the more general DFG to a DFT by creating several DFTs from one DFG. The splitting was done at nodes where the fan out was bigger than one. These points are also called common subexpressions (CSE). The compiler technique proposed in this paper attack the problem with SIMD instructions by permitting alternative solutions at each node. I.e. both optimal solution and also the possible solution with SIMD instructions. The decision of the instruction choice is postponed until later, when more information is available. When the whole DFG is covered, one tries to maximize the use of SIMD instructions. Of course there are certain constraints that must be satisfied for the code selection. 1. Every node can just be covered by one solution. 2. By selecting a SIMD instruction we have to make sure the children of the node also follow the SIMD instruction. For example, if we choose a SIMD instruction and the operand is stored in the lower part of 32-bit register, we have to ensure the result of the children nodes also store their results in the lower part of the register. 3. We also have to make sure that register usage at CSEs is consistent. I.e. when a node has an output that goes to more than one node, there might exist different coverage solutions.
4. We have to select SIMD instructions in pairs, no stand alone SIMD instructions. 5. No scheduling precedence between the SIMD nodes. 6. They have the same operator. 7. Input data and result must be located in the upper part of the register for one node and the lower part of the other node. 8. Data must be correctly aligned in the memory to ensure 32-bit loads and stores. 9. A node can only be covered by at most one SIMD instruction. 10. For a SIMD pair, if one of the nodes has to be scheduled before a third node and the other node within the SIMD pair has to be scheduled after the third node, we have a schedule deadlock situation and this is not allowed. The optimization problem is modeled as an integer linear program (ILP)

RESULTS
The algorithm has been targeted towards two different processors, TMS320C6201 from Texas Instruments and Trimedia TM1000 from Philips. The algorithm has been tested on different benchmarks, like vector add, IIR and FIR filter, and when it was possible to use SIMD instructions, the resulting code was more efficient. For the Trimedia processor, it was also possible to use SIMD instructions on 8-bit subregisters. The algorithm was able to explore this feature as well. The current limitations of the algorithm are that memory alignment and loop unrolling have to be done manually. The CPU times are also getting long for big DFGs. For embedded systems this might be OK, while code efficiency is of great importance. Another solution with the long CPU times is to split the DFG in to smaller ones.

CODE SELECTION FOR IRREGULAR DATA PATHS
As already mentioned in the introduction, the DSPs of today do not exhibit regular register files. Instead, certain registers are connected to functional units. You might also distinguish between operand registers and result registers. And of course, some registers are more general than others. For example some registers can only be input registers to the arithmetic unit and other to the multiplier. Then there are dedicated result registers, which in some cases can be input registers for other functional units. There might also be restrictions on the left and right input operand. This causes problems for the compiler, when it comes to register allocation. There will also be further restrictions on the register available when we try to execute instructions parallel.

While optimal code selection is possible by tree parsing in the case of DFT, it is NP-complete for the DFG case. A possible way to solve these kind of problems is by constraint logic programming (CLP). In the case of code selection for DFGs, a promising way is formulate the problem as a constrain satisfaction problem (CSP). All the variables in these kind of problems have constraints that defines mutual dependency between the different variables. Also, there is a certain domain associated with each variable. To solve this one simply map each variable to a value within its domain, such that the constraints are met. In the paper the use ECLiPSe to solve this kind of problems.

They present a new way of represent the coverage of the DFG. This is done through factored machine operations (FMO). An FMO is given by: \( \text{op}, \text{R}, \{O_1..O_n\}, \text{ERI}, \text{Cons} \). Op denotes an operation available on the processor, R is a result register, \( O_1..O_n \) are possible storage locations. ERI specifies further machine resources allocated by this machine operation. And finally, the Cons is the constraints associated with this operation. The first step in the algorithm is to cover all the nodes in the DFG with FMOs. There is also possible to have more than one FMO to each node by algebraic transformations. These must also be considered. We must also include the data flows between the nodes within the DFG. In order for output data to became input data in the next node, one might have to infer a register move or a memory storage. A DFG covered with FMOs reflects all the available alternative of machine operations at each node. The constraints ensure the selection of legal machine operations at each node. The constraints also make sure that there exists a legal path from the definition of a variable to it’s usage. One can show that the worst case complexity of FMO covering is \( O(N^4D^3) \). Where N is the number of nodes in the DFG and D is number of possible storage locations, registers.

The cost function in the algorithm is the cycle count. We also include the transfer costs of variables between nodes.
RESULTS
Four different compilation techniques has been simulated and compared. The target processor is a DSP from Analog Devices called ADSP-210x. First of all a traditional DFT covering model, secondly, DFG covering without allowing nodes with fan out out bigger than one to have chained instruction, like multiply-and-accumulate, and by third DFG coverage with chained instructions. Due to the large compilation times of big DFGs, there is also a modified variant. In this case is the DFG split into smaller DFTs. The code quality is of course better in the DFG covering case, but the compilation times becomes long. By splitting the DFG into smaller DFTs, the compilation times are reduced a lot, but the resulting code is still close to optimal.

CONCLUSION
The paper shows that by using the more general DFG instead of the DFT, we can get more efficient code. The method works well for the two different architectures tested in the paper. First on a SIMD architecture and than on a DSP with irregular data path. On the negative side is the long simulation times for large DFGs. One way to reduce these long times is to split the existing DFGs into smaller ones. The resulting code is still close to optimal.