Report on

Software Synthesis

Submitted to Prof.Anshul Kumar

by

A.S.Pramod Kumar 2001MCS027

S.Divakar Reddy 2001VLS014

Abstract

Software synthesis is the task of converting a complex executable specification into a traditional software program that can be compiled by traditional compilers. It mainly consists of 2-sub problems a) Task Handling and b) Code generation. The 2 sub problems are explained with emphasis on code generation. Task handling deals with the task of generation of source code keeping in view of the timing constraints and the typical runtime support which involves multi tasking, process communication and process synchronization. Code generation deals with the code optimization and code generation techniques (back end of the compiler) in the case of retargetable compilers. Code generation mainly consists of 4 phases namely Code selection, instruction scheduling, register allocation and address code generation.

Introduction:

Software synthesis is a new approach that focuses on the support of embedded systems using customized kernels and compilers different from the traditional ones, which can exploit the special architectural features of the embedded processor and the application, which is known a priori. Compared to traditional design practices, a better utilization of the available time and hardware resources can be achieved with software synthesis, because the static information provided by the system specification is fully exploited.

Motivation:

- **Exploring the ASIP design space**

  ASIPs are a compromise between domain-specific processors and non-programmable ASICs. ASIPs are still programmable, but they serve only a very narrow range of applications. Sometimes ASIPs can be parameterized: In this case, the basic architecture of an ASIP is fixed, but it can be customized for a given application by setting a no. of different parameters. In this way, word lengths may be adjusted to the required precision, register files may be sized and available special hardware components, e.g. hardware loop support may or may not be included. Since these parameters are mostly orthogonal to each other, a large no. of compilers are used. In order to avoid this effort, retargetable compilers are used capable of generating code for any particular ASIP configuration.
- **Efficient Code Generation**
  - **Area**

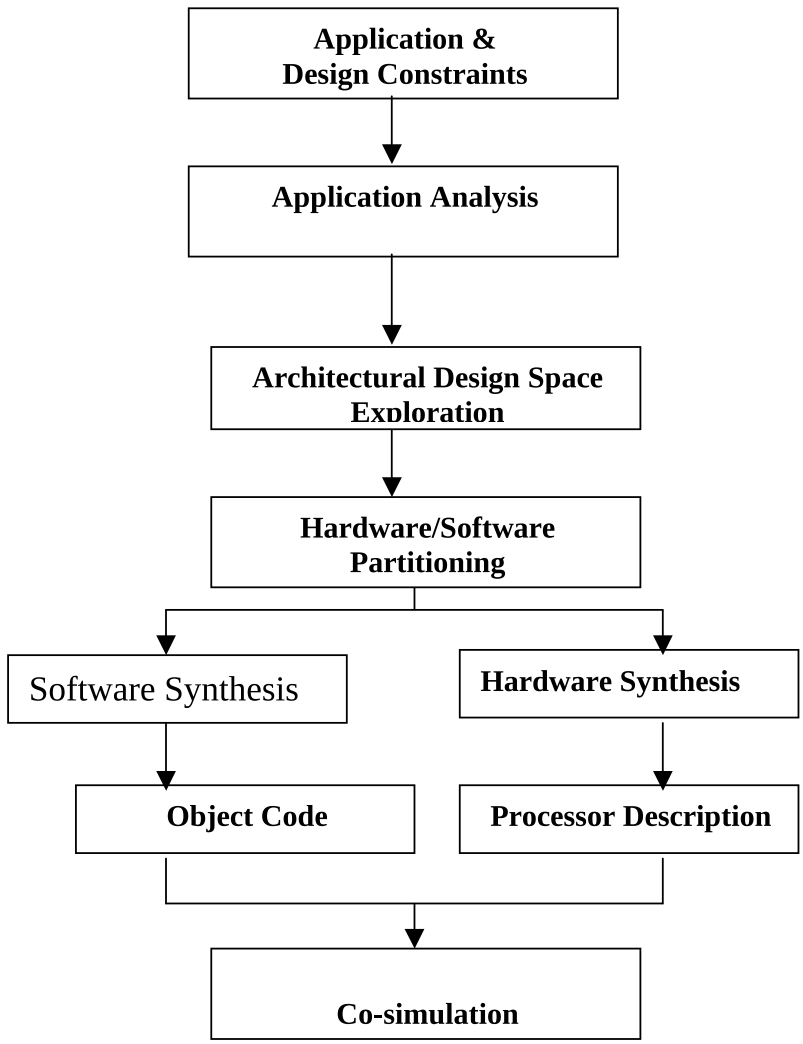
    In embedded systems-on-a-chip, where program memories are integrated with cores, a larger code size immediately implies a lower chip yield and thus higher costs.

  - **Performance**
  - **Power**

The program code should be placed in memory such that successive accesses to memory access consecutive locations so that the memory transitions on the bus is less which reduces the power. Power is of great concern in embedded systems because most of them are driven by battery.

- **Code Optimization**
- **Retargetability**

Design Methodology:



Once the application is given along with the cost, time and power constraints, it is analyzed and then hardware/software partitioning as well as software, hardware and interface synthesis is done. Based on that object code and processor description is obtained. Using this information, co-sim8ulation is done to check whether the constraints are met. If they are met, then we stop the process otherwise we go back to the partitioning phase and redo the entire process until the imposed constraints are met.

More in particular, hardware/software partitioning is supposed to be already performed, so only portions of the system specification assigned to the software partition are considered.
nics and personal communication systems. The distinctive characteristic of these systems is the coexistence of signal

processing and control functions, which require the support of different kinds of timing constraints:

- Signal processing functions operate on sampled data streams, and are subject to real-time constraints derived from the required sample frequency (throughput) and latency, as determined by the environment.

    - Control procedures may vary in nature. On the one hand, soft deadlines may be imposed (e.g. in a man-machine interface). In this case the procedure has to be executed as soon as possible, but an occasional delay in the execution does not compromise the integrity of the entire system. On the other hand, hard deadlines occur (e.g. in a critical feedback control loop). In this case the timing constraints are very stringent.

Traditionally, specialized operating systems called real-time kernels are used for the software support of complex systems. Software synthesis, as discussed in this chapter, is an alternative approach to real-time kernels: starting from a system specification, typically composed of concurrent communicating processes with timing constraints, the aim of software synthesis is the automatic generation of the source code, realizing: (1) the specified functionalities while satisfying the timing constraints, and (2) the typical run-time support required for real-time systems, such as multi-tasking, and the primitives for process communication and synchronization. Compared to real-time kernels, a better utilization of the available time and hardware resources can be achieved with software synthesis, because the static information provided by the system specification is fully exploited. The automatically generated run-time support is customized for each particular input specification, and does not need to be general, as in the case of real-time kernels. Moreover, an accurate static analysis, usually performed before software synthesis, provides an early feedback to the designer on the feasibility of the input specifications. In this way the trial and error design cycle, typical for real-time kernels, is avoided. Finally, since the output of software synthesis is source code, portability can be easily achieved by means of a retargetable code generator. This is especially interesting when the target processor is still subject to changes; like in the case of application specific instruction set processors (ASIPs).

APPROACHES IN SOFTWARE DESIGN

Real-time kernels: Real-time kernels have been used extensively as a software support in the design of embedded systems. Most of these kernels are stripped versions of a traditional time-sharing operating system, made appropriate for the real-time domain by reducing the run-time overhead. These small kernels, often with limited functionality, are in the first place designed to be fast by ensuring a fast context switch, a quick response to interrupts and a minimal interrupt disable time.

Above all real-time kernels provide: (1) the run-time support for real-time multi-tasking to perform software scheduling, and (2) the required primitives for inter-process communication and synchronization, and for accessing the hardware resources (typically through an application procedural interface). Since processes are considered as black boxes, most kernels apply a coarse grain model for process scheduling. Usually a fixed-priority preemptive scheduling mechanism is used, where process priorities have to be specified, rather than timing constraints. Assignment of process priorities, as in the case of the fixed priority-scheduling scheme, is a manual task to be performed without any tool support. Typically, an iterative and error-prone design cycle, with a lot of code and priority tuning, is required. Not only is this approach inflexible (adding one task can cause the cycle to be re-iterated) and time consuming, but it also only guarantees correctness for the selected (simulated) stimuli. Additionally, the behavior of the scheduler under peak load conditions is hard to predict, resulting often in under-utilized systems to stay on the safe side

Alternatively, traditional scheduling approaches use timing constraints, specified under the form of a process period, release time and deadline. From the designer viewpoint however, these constraints are more naturally specified with respect to the occurrence of observable events. Moreover, the scheduler has no knowledge about the points in time when the events are generated by the processes, and consequently cannot exploit this. It is safer to guarantee timeliness at pre-runtime, as new family of kernels tends to achieve. Moreover, kernels trade optimality for generality, resulting in a significant runtime and memory overhead.

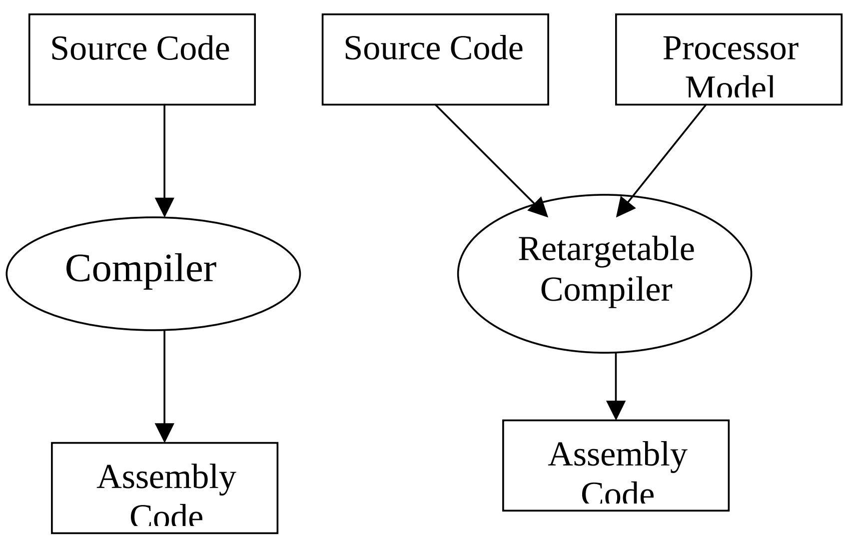
On the other hand, static scheduling techniques can be used efficiently. In practice this is however only applicable to periodic processes, while asynchronous processes introduce significant overheads, such as active waiting.

One of the earliest approaches for hardware-software co-design is the Vulcan system. In this system software synthesis is performed in conjunction with hardware/software partitioning: as a starting point program threads are extracted from the system specification, composed of concurrent processes. This step is done in order to isolate operations with an unknown timing delay (ND-operations) at specific places, namely at the beginning of the program threads. Initially all operations are supposed to be performed in hardware, except for all ND-operations; partitioning is then performed by means of an iterative approach which selects operations to be transferred from the hardware to the software partition, in order to minimize the hardware cost, while satisfying the imposed timing constraints. The run-time behavior of the software partition is controlled by a run-time scheduler, which alternates the execution of the program threads in order to achieve the original process concurrency. The run-time scheduler activates the threads based on a simple control-FIFO containing pointers to the threads which are ready to run. The next thread to be executed is pushed on the control-FIFO directly by the threads under execution, while the run-time scheduler simply pops the program thread stored on the top of the control-FIFO and executes it, without performing any additional ordering of the threads, which are ready to run. This simple scheduling scheme provides only a restricted support for timing constraints. Moreover interrupts are not supported, due to the choice of using a non-preemptive scheduler.

CODE GENERATION

Def: A compiler is called retargetable, if it can be modified to generate assembly code for different architectures, while reusing the largest part of the compiler source code

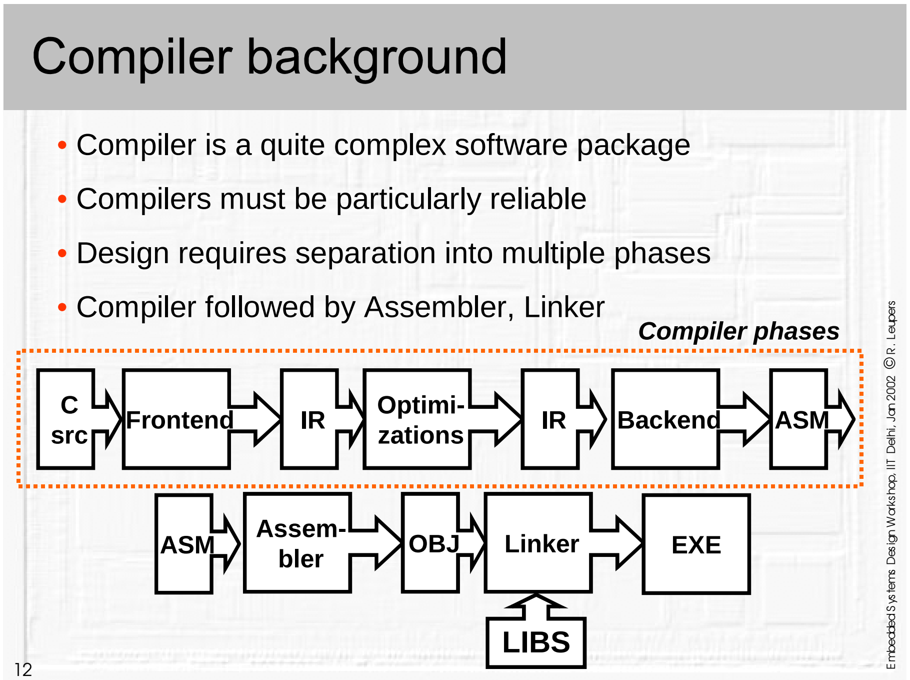
Classical Compiler Retargetable Compiler



A compiler is called retargetable, if it has a

- Single tool for an entire class of target machines
- Support for design space exploration by editing the target processor model
- Tradeoff retargetability/code quality: Realistic retargetable compilers focus on a specific class of processor architectures

Degrees of retargetability:



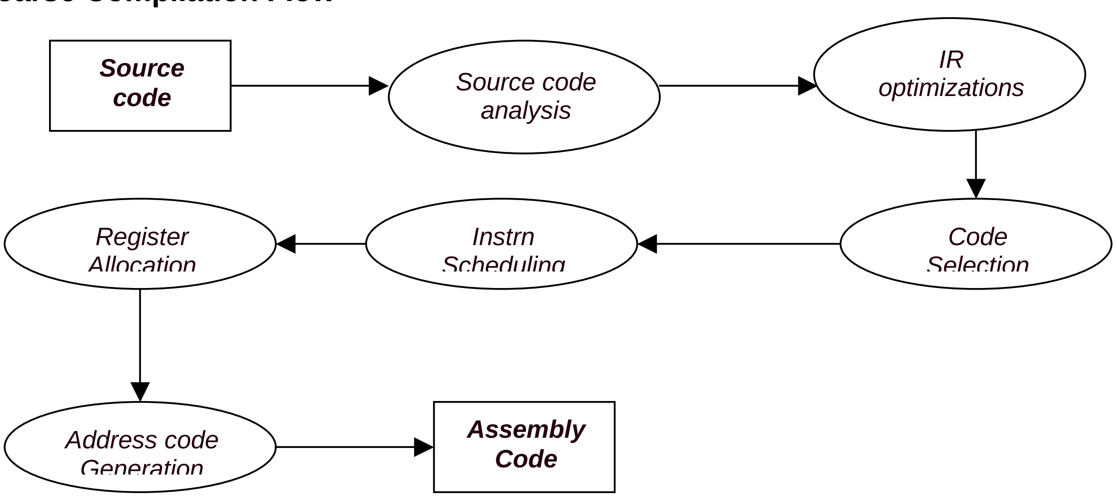
■ Portable: Some source code can be kept, but a lot of code needs to be newly written

■ Developer retargetable: Compiler developer (or very experienced user) can retarget the compiler with limited effort

■ User retargetable: User without in-depth compiler knowledge can change the target

■ Parameterizable: Source code is fixed, retargeting mainly by adjusting several parameters

Coarse Compilation Flow



IR Formats

•Statement Oriented Formats: Three Address Code
  Simple, suitable for IR optimizations

•Graph Oriented Formats:CDFG
  Expressive, suitable for code generation

Machine Independent IR Optimizations•**Common sub expression elimination**: IA computation is redundant at some point of the program, if the computation is already been performed at an earlier program point. Such redundancies can be eliminated by keeping the result of the first occurrence of the computation in the register and reusing the register contents.

•**Dead code elimination**: Any computation that generates a result which is guaranteed to be never used again in the program can be eliminated.

•**Loop-invariant code motion**: Computations contained in the loop, whose results do not depend on other computations performed in that loop are loop invariant and can thus be moved outside the loop code in order to increase performance.

•**Constant folding:** Computations that are guaranteed to result in constants and be moved from program run-time to compile-time. In this case, the compiler folds the computations on constants to single constants already when generating assembly code.

Mapping of machine independent IR into machine specific assembly instructions has 4 main phases

- *Code selection*
- *Scheduling*
- *Register allocation*
- *Address code optimization*

Code Selection:

- Decide which assembly instructions will be used to implement IR statements
- Does not work on statement-by-statement basis because no one-to-one correspondence between IR statements & assembly instructions.
- Goal to optimize set of assembly instructions whose accumulated cost is minimal.
- Done using tree parsing (Takes input a DFT)

Instruction Scheduling

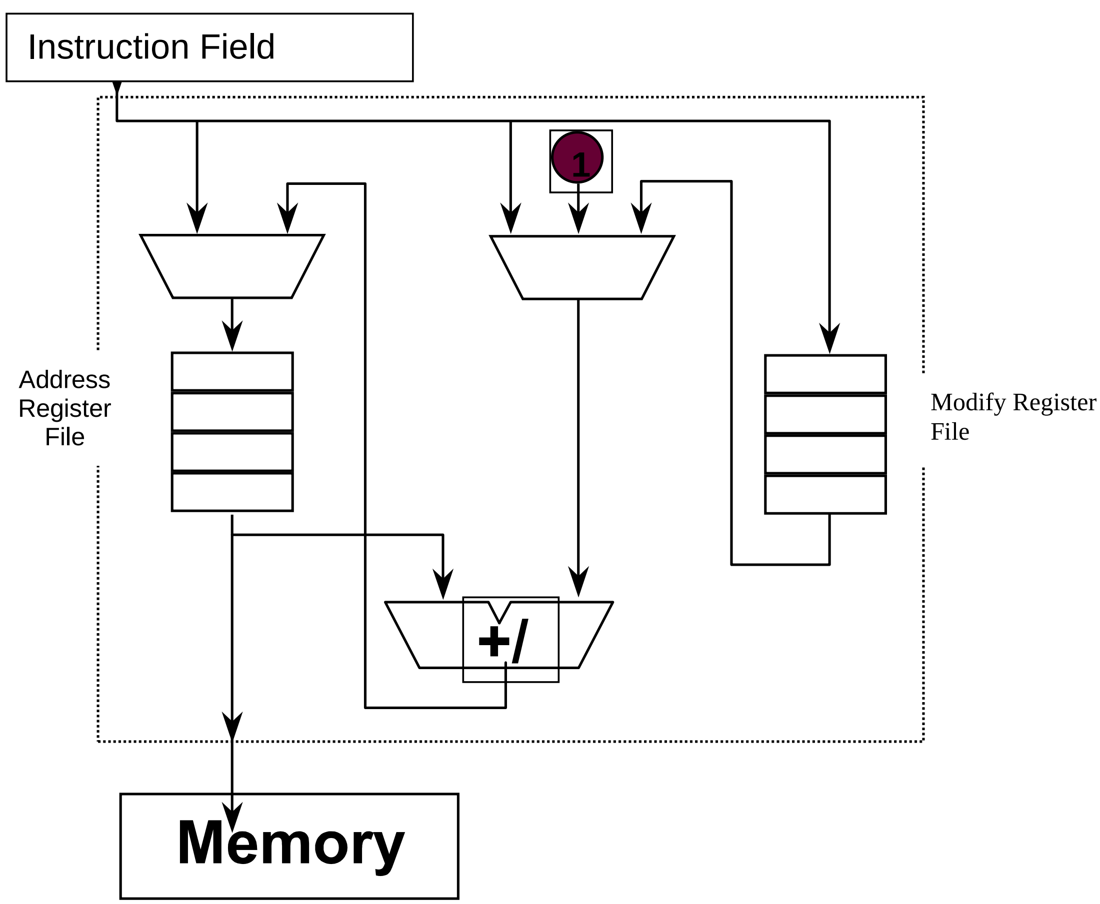
- Decide which assembly instructions will be used to implement IR statements
- Does not work on statement-by-statement basis because no one-to-one correspondence between IR statements & assembly instructions.

- to optimize set of assembly instructions whose accumulated cost is minimal.
- Done using tree parsing (Takes input a DFT)

Register Allocation for DSP Data Paths

•Problem of assigning virtual registers to a fixed no. of physical registers while generating spill code in case that physical limit can't be met.
•Represented as interference graph.
•Each node represents one virtual register.
•Each edge represents lifetime overlap between pair of registers.
•The interference graph represents a relation that two virtual registers must not be mapped to the same physical register.
•Call physical register numbers as colors and there are k registers available, the register allocation is the task of coloring the interference graph with at most k colors in such a way that adjacent nodes are assigned different colors.

Generic address generation unit (AGU) model



In general DSP's offer mainly two modes:

- Direct: The direct addressing mode uses an immediate field in the instruction word to form memory addresses.
- Indirect. Addresses are read from address registers (AR's) .

More complex modes like base+offset addressing which are common for CISC are not provided. Instead, DSP's permit address to be computed in parallel to the central datapath. This means that a memory address required in instruction cycle n sometimes can be computed by an AR modification in cycle n-1 without the need for extra instructions. Such parallel address computations are supported by a dedicated address generation unit (AGU).

The AGU comprises a file of k address registers, as well as a file of m modify registers (MRs). AR and MR indices are provided by AR and MR pointers, which are either part of the instruction word or are stored in special registers. For a given AR pointer value p, AR[p] is the current AR holding the effective address for memory access in indirect mode. According to the register transfer, there are following ways of computing address by modifying an AR in the AGU.

Immediate load: A constant c is loaded from an immediate instruction field into the current AR.

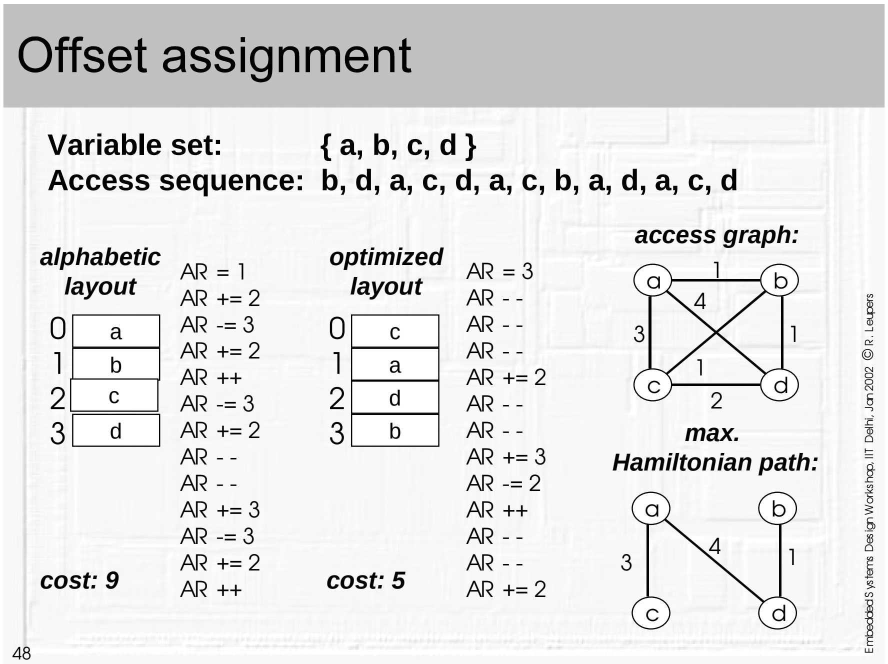
Immediate modify: A constant c from an immediate instruction field is added to (or subtracted from) the current AR.

Auto-increment: The current AR is incremented (or decremented) by a constant d. The absolute value of d is bounded by some "small" constant r.

Auto-modify: The current AR is incremented (or decremented) by the contents of the current modify register MR for some value q of the MR pointer.

The only address computations not requiring an immediate constant are auto-increment and auto-modify. These are also the only address computations that can be executed in parallel to the other instructions. The reason is that encoding of the short immediate constant d does not require an extra instruction word. If b bits are available for encoding d, then d is bounded by r=pow(2,b-1). Auto-modify operations do not require any immediate constant at all since an MR can be modified implicitly.
All other address computations consume an extra machine instruction and thus increase code size and reduce performance. For instance, an immediate modify is required, whether the modifier c is larger than that r, so that the c has to be encoded in a separate word. Therefore, the goal in address code generation for DSPs must be to maximize the use of auto-increment and auto-modify operations for generating address.

Offset Assignment



In the fig above, the variable set and access sequence are shown below. The variables should be places in the memory such that the accesses are mostly possible with auto increment/decrement operations, which cost 0.Rest of the operations cost 1.The goal is to reduce the cost of accessing the operations.

In the first ordering, the cost is 9.In the second ordering the cost is 5.
The access graph is built using the access pattern and the maximum weighted Hamiltonian path is selected. The variables are placed following that order.

Address Register Allocation for Array accesses

Accesses are represented as nodes.
Accesses within range of ‘r’ are connected by edges.

The problem of address register allocation for array accesses is reduced to finding disjoint sets (k) such that within each set, array accesses fall in the range of r and so can be allocated one AR.
This can be solved by using minimum disjoint cover problem.

Function inlining: advantages and limitations

Another source level optimization is function inlining so that we can avoid the overhead of pushing all the registers into the run-time stack before branching to the function.
But this increases the source code size. In case we have more chip area, we an improve the performance by function inlining. Selecting the candidates to be inlined should be done keeping in view of many factors. Otherwise there will be an explosion of code size.

Advantage: low calling overhead

```
 push PC;
push b;
BRA sq;
 pull R1;
 mul R1,R1,R1;
 pull R2;
 push R1;
 BRA (R2)+1;
pull R1;
ST R1,a;
```

branching

```
Function sq(c:integer)
 return:integer;
begin
 return c*c
end;
....
a=sq(b);
....
```

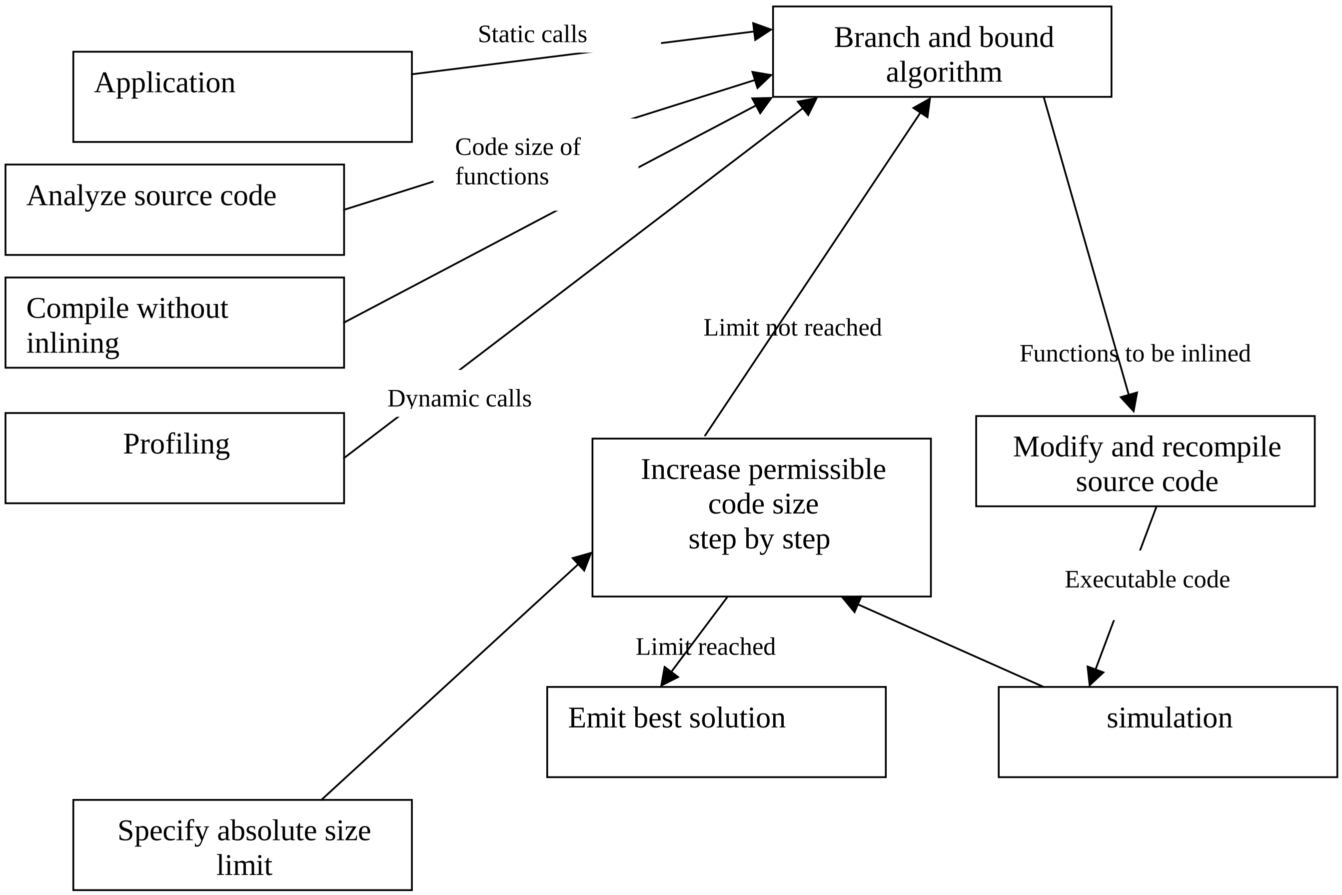
Inlining

```
LD R1,b;
MUL R1,R1,R1;
ST R1,a
....
```

Limitations
- Not all functions are candidates.
- Code size explosion.
- Requires manual identification using ‘inline’ qualifier

Our goal:
- Controlled code size
- Automatic identification of suitable functions

Design flow


Application
Analyze source code
Compile without inlining
Profiling
Static calls
Code size of functions
Dynamic calls
Branch and bound algorithm
Limit not reached
Functions to be inlined
Increase permissible code size step by step
Modify and recompile source code
Executable code
Limit reached
Emit best solution
simulation
Specify absolute size limit

Conclusion

- The compiler techniques used for general purpose processors can’t be used directly for embedded processors, which end up in generating inefficient code.

- The embedded processors show special architectural features, which can’t be exploited with standard compiler techniques.

- Retargetability gives us the flexibility, which can be used in architecture design space exploration and generation of efficient code.

References

- Rainer Leupers , *Code optimization techniques for Embedded Processors ,2000*
- P. Marwedel , *Code generation for embedded processors,Kluwer 1995*.
- Ashok Sudarsanam , *Code optimization libraries for retargetable compilation for embedded digital signal processors.1998*
- ISDL: *An Instruction set description language for Retargetability*.
- Macro cornero,Filip Theoen,Gert Goossens and Franco Curatelli, *Software synthesis for real-time information processing systems*.
- Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong, *Specification and Design of Embedded Systems*

Ongoing Projects on Retargetable Compilers

- GNU C compiler
- Little C compiler(Princeton Univ)
- SUIF(Stanford Univ):Optimizing frontend for C/C++
- LANCE (Univ of Dortmund):Optimizing Ansi C Frontend
- OLIVE code generator
- SPAM :Retargetable Compiler for DSPs
- CHESS :Retargetable compiler for DSPs
- Trimaran(HP Labs):Retargetable compiler for VLIW
- MSSQ (University of Kiel/Dortmund):Retargetable Compiler for ASIPs