Developing a simulation platform for an experimental architecture

BART GOEMAN, KOEN DE BOSSCHERE and HENK NEEFS
Department of Electronics and Information Systems (ELIS)
Universiteit Gent
St.-Pietersnieuwstraat 41, B-9000 Gent
BELGIUM

Abstract: - In this paper, we describe how we have bootstrapped a compiler/simulator chain for an experimental computer architecture. The problem was that in the beginning, there was neither a compiler, nor a simulator, and that both had to be developed simultaneously, while the architecture was still evolving over time. We describe how the software was created in two stages: we started with a functional platform, derived from the existing SimpleScalar toolset, and this functional version was then used as starting point for the real architectural platform. The functional compiler was used to develop the functional simulator, which was in turn used to develop the optimizing compiler and the architectural simulator. In addition, we describe the five techniques that had the biggest impact on the development speed of the compiler and simulator: the development of a functional compiler by means of post-processing, the use of a reference execution to verify the simulator operation, an text-based interface between the compiler and the simulator, an object oriented design, and ample trace-based debugging facilities.

Key-Words: computer architecture, simulation, Block Structured Architecture *CSCC'99* Proc.pp..3681-3688

1 Introduction

Simulation is a cheap and powerful technique during the design stage of a new computer architecture. It does not only allow for a cost-effective and quick evaluation of new micro-architectural concepts (e.g., new branch prediction schemes, memory hierarchies, the effect of speculative execution), but it also allows to experiment with the visible aspects of the architecture, such as the number of registers, the availability of particular instructions, the word size, etc. In the latter case, the simulator will require different program code to work on. Hence, an extra set of companion tools that match the simulator is required (compiler, assembler, linker and libraries). We will call the combination of the simulator and the set of companion tools the *simulation platform.* A simulator in which some of the visible architectural parameters can still be varied is called a *simulator for an experimental architecture.*

One of the major problems with the development of a simulation platform for a new experimental architecture is that both the simulator and the companion tools must be developed in parallel, which creates a bootstrapping problem. The first version of the simulator must be developed without the availability of *real programs*, such as the Spec benchmarks, and the first version of the compiler must developed without simulator to check the correctness of the generated code.

Since developing a complete simulator or compiler is a non-trivial task that requires several thousands of lines of code to be written, the lack of a minimal simulator or compiler is seriously slowing down the process of debugging the complete platform. A major difficulty is that the real cause of a bug can be hidden in either the compilation tools or in the simulator, and that sometimes it is very hard to track it down. In order to efficiently debug the simulation platform, it is therefore of paramount importance to detect the real cause of a bug as soon as possible.

In this paper we present the approach we have used in developing a simulator for an architecture under development, the block-structured architecture (BSA) [2], in which instructions are grouped in fixed-size blocks, and in which the instructions that belong to a single block are executed in a data-flow manner. Instructions in a block communicate by exchanging values along the data flow paths, blocks exchange values by means of a set of registers. Since this is an experimental architecture, the size of the blocks, the number of registers, the internal subdivision of a block (ALU instructions, control flow instructions, data movement instructions), and even the instructions themselves must be modifiable. Changing these architectural parameters however implies that the compiler must have the ability to generate code that can be substantially different, and the simulator must be flexible enough to cope with it.

In order to efficiently develop the simulation platform we decided to make maximally use of existing tools and code, and to make sure that errors in one of the tools of the platform could be tracked down as soon as

possible. In order to reach this goal, we have developed two platforms. A first 'functional' platform was based upon existing tools (SimpleScalar toolset [1]). This platform was able to compile and simulate the Spec benchmarks, albeit neither in a very optimized way, nor was it able to compute cycle-true execution times. Section 2 describes the functional platform. Section 3 describes the second 'architectural' platform, which was bootstrapped from the functional platform, generates more optimal code, and contains an accurate simulator for the BSA (including e.g., out-of-order and speculative execution).

In other words, we started with a preliminary (called functional) version of the compiler that enabled us to thoroughly test the functional simulator. Once the functional simulator was stable, it was used to develop the optimized compiler and the architectural simulator. The functional version of the compiler was then no longer used.

The remaining sections contain an evaluation of our approach, some related and future work, and a conclusion.

2 Functional platform

The basic functionality of the functional platform is to functionally execute a program. This means that the instructions will be executed in the same order in which they occur in the program, without the performance enhancing techniques that are currently available (out-of-order execution, speculative execution, etc.). The only interest of a functional simulator is to obtain the result of a given program. This in contrast with an architectural (also called cycle-true) simulator which provides much more information, and is generally not used to obtain the outcome of a given program, but instead to learn more about the execution of the program. We describe here how we bootstrapped the functional platform from the SimpleScalar toolset.

2.1 A post-processor as code-generator

Writing a full compiler for a new architecture is generally not needed these days. Since all relevant benchmarks that are used to evaluate an architecture are either written in C or in Fortran (Spec benchmarks), and since Fortran can efficiently be translated into C (e.g., by means of the `f2c` translator), it is generally sufficient to implement a C-compiler. Since there are several publicly available C-compilers around, it often suffices to retarget the compiler, i.e., to add a new code generation stage to the compiler. Some compilers (e.g., gcc) have a generic code generator that can be customized by means of an accurate machine description.

The 'functional' compiler for the BSA even went one step beyond retargeting in the sense that it even reused the back-end of the SimpleScalar compiler. The instruction set of our BSA turns out to have many instructions in common with the SimpleScalar-architecture. So, we took the complete SimpleScalar compiler, and we wrote a post-processor to translate the code it produces into BSA-code. The post-processor does a source-to-source translation of the assembly language output of the SimpleScalar compiler. Hence, its output is a readable BSA-assembly language program that can be visually inspected for correctness

This approach results in non-optimal but correct code (we basically create one BSA-block per basic block, hence, the functional correctness of the output will entirely depend on the correctness of the input which we trust). Carrying out sophisticated BSA-optimizations in a post-processor would be difficult anyhow because important information, such as aliasing information about variables is not available anymore at this level. The major advantage of the post-processor approach for this first compiler version is that it can be implemented fast. The final optimized code-generator for the BSA is developed for the second, architectural, platform.

2.2 The use of a reference execution

The fact that our compiler produces BSA-code that can be related with the original SimpleScalar code has had an important application in the initial debugging of the simulator and has considerably reduced the time needed to debug the simulator.

The idea is as follows: since both programs are computing the same result with very similar instructions, many of the BSA-instructions must produce exactly the same result as the corresponding instruction in the original program. By asking the post-processor to add information about the original SimpleScalar instruction to every instruction of the BSA-program, the BSA-simulator has all the information it needs to create a link between the two executions. However, if an instruction manipulates a value that is an instruction pointer (e.g. return address), the corresponding BSA-instruction manipulates a block pointer; these values are not the same. The simulator has no means to determine if a certain value is an instruction pointer or not. But in all cases, both programs follow the same control path. In order to avoid false warnings, only the control path is checked.

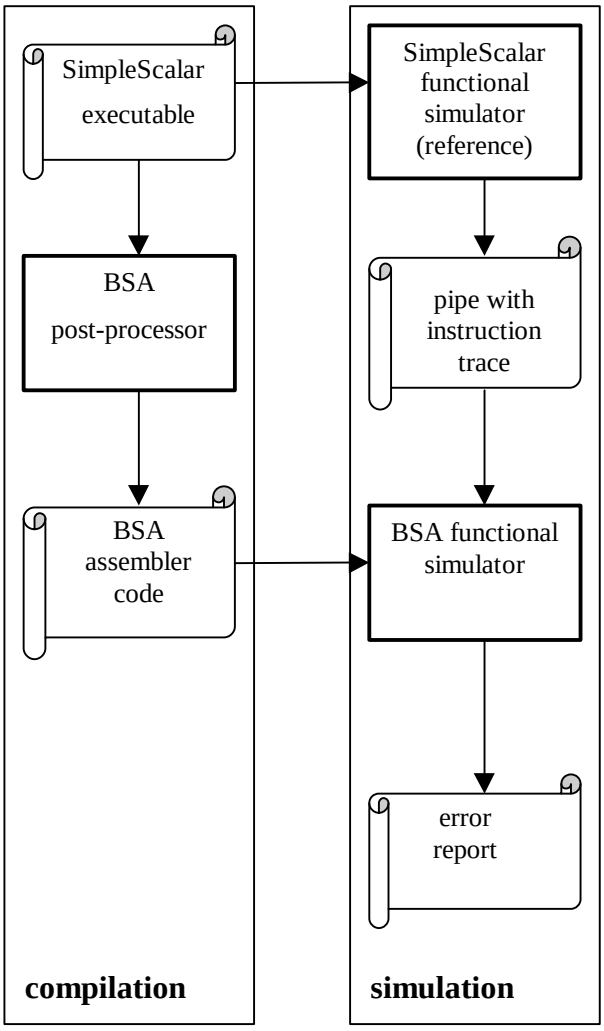

Figure 1: The reference mechanism

As program traces are usually huge, it is not feasible to store them. Instead, the BSA-simulator forks off the SimpleScalar simulator on the corresponding program, and both simulators communicate by means of a pipe. The verification of the execution of the BSA-simulator against a reference guarantees that an error made by the BSA-simulator is detected either at the end of the block, or shortly afterwards.

This facility has been proved to be extremely useful for debugging the BSA-simulator. It furthermore made us confident that the BSA-simulator works correctly. Not only do we see that the Spec-benchmarks produce the correct results, but on top of that, we also know that all internal operations that can be related to the original program are also correct, even in the case where for some reason, they do not directly affect the output. We were even able to find some errors in the SimpleScalar simulator.

2.3 A text-based compiler-simulator interface

The interface between compiler and simulator is not a binary executable, but a plain text file with the assembly language source program, enhanced with the relation between the BSA-instructions and the SimpleScalar instructions they originate from, and some other directives to the simulator.

There are several reasons why we use a source file as input to the simulator:

1. It makes visual inspection of the simulator input straightforward without extra tools. It also allows the developer to modify the input program by means of a simple editor for testing purposes.

2. It makes the binary encoding of the instructions unnecessary. Remember that we are creating a simulation platform for an experimental architecture. Changing architectural parameters like the number of registers might require the design of a new binary format, and some parameter values might even be excluded because they are not compatible with a given binary format (e.g., it is hard to code three register operands in 32-bit instruction words in an architecture with e.g., 1024 registers). A source representation of the program does not have these limitations, and is therefore better suited for an experimental architecture.

3. The simulator front-end that reads in the program can easily be generated by means of lex and yacc. The development of such a front-end is no wasted effort, because in the final platform, it can be re-used as the front-end of the assembler. In the first platform, we are actually merging the assembler and the simulator. The front-end also checks the code thoroughly. Invalid instructions are detected in an early stage.

4. A direct link between the compiler output and the simulator enables us to communicate extra information between the compiler and the simulator. This information can be added to the compiler output in a variety of ways without having to worry how this information must be represented in the binary format. Currently, there are two extra sources of information that are being communicated this way: (i) the relation between the original SimpleScalar instructions and the BSA-instructions. This information is necessary to verify the operation of the BSA-simulator against the operation of the reference simulator; (ii) the definition of the architecture. Since we are developing a simulator of an experimental architecture, it is important that the compiler communicates the relevant parameters of

the architecture (e.g., number of registers, word size, structure of a block, etc.) to the simulator. If this would not be the case, the simulator could use the wrong architecture, and hence produce strange errors. The fact that this information is contained in the program was of great help during the development of the simulation platform. The simulator never ran with the wrong architectural parameters.

2.4 Object oriented design

From the very beginning we decided that the simulation platform had to be designed using an object-oriented methodology. The flexibility that comes with an object-oriented implementation is extremely useful when studying an experimental architecture. Indeed, by modeling every functional part of the architecture as an object, and by letting objects communicate by means of messages, it is very easy to conduct experiments by replacing a particular component in the simulator. Once that the object interface to e.g., a branch predictor has been defined, it can be re-implemented easily to play with several different prediction schemes. Since the interface with the simulator is separated from the implementation, even students that do not fully understand the implementation of the simulator, can replace it with their own branch predictor. Many of the hard to solve problems during the development of the simulator were caused by non-object oriented parts that were copied from the SimpleScalar simulator which is written in C.

The fact that the object interfaces were clearly defined, created yet another opportunity to prevent errors from hiding themselves deep in the simulator. We took advantage of the object interfaces to explicitly check all the arguments passed to objects. As soon as some of the arguments are out of range or contain garbage, an error message is produced. This allowed us to even find a bug in one of the spec benchmarks where we discovered that one of the stack cells that were used by *cc1* was not initialized, and hence contained garbage. Fortunately, the value read was not used by the program, and hence, did not cause any harm.

2.5 Trace-based debugging tools

Although we did everything to prevent bugs from occurring, we still needed some tools to efficiently find remaining errors in the program. We did not provide support for interactive single stepping or breakpoints. Instead, we added an equivalent tracing mechanism that allows us to trace the execution of the program at various levels of detail, and the possibility to take snapshots of the simulator state. The choice for tracing was motivated by the fact that a simulation run often takes hours to finish. It does not make sense to wait for hours in an interactive debugger until a particular error in a program occurs. An error is frequently discovered by the symptoms it causes, but when the symptoms occur, it is often too late to find out what caused them, and one has to restart the execution to track down the real cause of the error (cyclic debugging). Hence, for debugging such a long-running program it is better to produce a detailed trace of (part of) the execution. Analysis of a detailed trace is then a more effective way to find the error as it will require less reruns. A second observation was that a debugger also takes resources, and hence slows down the simulator even further. Therefore, we decided that an interactive approach was impractical.

2.5.1 Changing the trace-level

The major impediment when tracing a program is the huge size of the resulting traces. Therefore, we implemented the following mechanism. A user can add special trace commands (actually, this is in the form of comments) to his program (this is easy because the program is available in source format). These trace commands will switch on and off the tracing when particular events occur. Besides switching the tracing on and off, they can also change the kind of information and amount of information collected in the trace (several different trace levels). This allows the developer to really track down the error after a few runs of the program.

2.5.2 Taking snapshots

By means of the same mechanism, the developer can also add commands that store a snapshot of various parts of the state of the simulator (e.g., a part of the main memory, the registers, the complete state of the processor core, part of the stack). All information that is necessary to interpret this information (e.g., call chains) is also provided. In addition to this, the developer can also request that a subset of the objects the simulator is composed of, dump their state on the trace file. This feature has also helped quite a lot to efficiently debug the first versions of the simulator.

2.5.3 Post mortem debugging

If this information is insufficient to track down the error, the simulator can be forced into a core dump at the moment the error occurs or when the simulator crashes. This provides access to all data structures by means of a standard debugger, amongst which the sequence number of the block that was being executed. This number can be used in a subsequent run to start detailed tracing e.g. 100 blocks before the problematic block would be executed again. This feature makes it rather straightforward to track down the real cause of a crash. It turned out that most errors could be fixed in just a couple of runs of the simulator.

3 The architectural platform

Once the functional platform was stable, it was used to bootstrap the architectural platform, composed of a cycle-by-cycle out-of-order simulator and an optimizing compiler. The same compiler-simulator interface, object-oriented design, and debugging tools were used, but the compiler and reference execution were different.

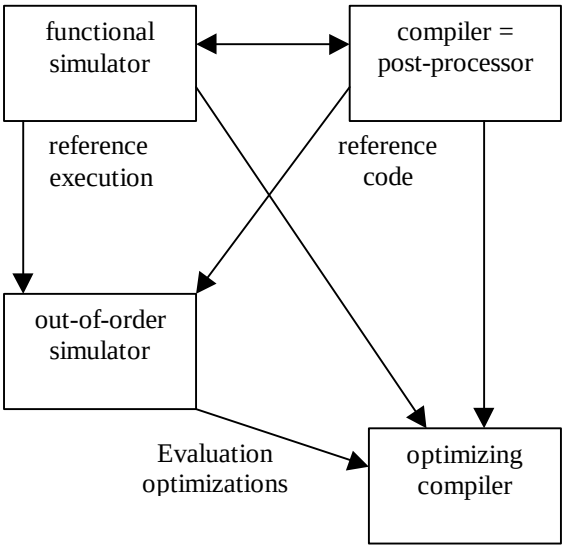

Fig. 2: bootstrapping process. The arrows indicate the dependencies (during development)

First, an out-of-order simulator is built. Testing is done using the trusted code from the post-processor. The result can be compared with the output from the functional simulator.

Second, an optimizing compiler is built. This one uses the functional simulator and the post-processor to check for correctness. The out-of-order simulator is used to measure the impact from these optimizations on performance.

3.1 Optimizing compiler

This compiler has to carry out optimizations that are essential for a BSA: superblock and hyperblock formation, i.e., the formation of large basic blocks by means of predication, in-lining, code duplication, etc. These optimizations require information from the compiler that is not available anymore in the SimpleScalar output, and therefore the optimizing compiler is now no longer designed as a post-processor for the SimpleScalar output, but as a compiler back-end in its own right. The interface between compiler and simulator is still a text file, but this time, the relation between the optimized and the original SimpleScalar code has completely disappeared. This is however not a problem anymore since we now trust the functional simulator (it has been tested thoroughly during the development of the functional platform). The reference execution was only necessary to debug the functional simulator, not to use it.

3.2 The reference execution

For the architectural simulator, we again try to rely on a reference execution to speed up the debugging process. This time, the reference execution for the architectural simulator is provided by the functional simulator executing the same optimized code. Since block numbers in the functional and architectural simulator are identical (after all, they are executing the same program), the relation between the functional and architectural execution is now fairly straightforward.

This is however not true for the instructions in a block. The architectural simulator uses an out-of-order model for the execution the instructions in a BSA block. Forwarding, branch (mis-)prediction, data value speculation are modeled explicitly, resulting in a different instruction stream compared to the functional simulator (this time it is a data flow execution instead of a sequential execution). However, the blocks themselves are executed sequentially. After finishing a block, both simulators should be in the same state, which can be compared. Hence, we can compare the values produced by the blocks, and the control path between blocks.

The functional simulator is forked off by the architectural simulator and communicates values and control flow information to the architectural simulator through a pipe. As a result, it is impossible that an error remains unnoticed longer than one block (containing some tens of instructions). Debugging this simulator was an easy job.

4 Evaluation

Since we tried to maximally reuse existing code and tools, we succeeded in implementing the functional and architectural simulation platform in about 10 person months by two (inexperienced) undergraduate students (the optimizing compiler is not included in this effort, as it was not included in the initial effort to produce the architectural simulation platform). Both simulators and the post-processor are approximately 30000 lines of C++ code. The code that was developed is able to correctly compile and simulate the eight SPECint95 benchmarks, and five Unix utilities (wc, grep, perl, yacc and lex). This implied that a minimal OS-API had to be implemented in the simulator too. This aspect was taken care of by reusing existing libraries from the SimpleScalar simulator.

Although execution speed was not a major

concern — the priorities were: correctness (accuracy), flexibility, maintainability, robustness — we give here some information on the simulation speed.

Table 1 shows the simulation time for yacc and lex, and this for both BSA simulators (functional and architectural) and their SimpleScalar (SS) equivalents. The simulation time is also compared to a native version of the program.

	unit	yacc	lex
#SS Instructions	10^6	4.47	39.45
Native	s	0.06	0.16
SS functional	s	2.77	22.8
	slowdown	46	143
SS architectural	s	67	541
	slowdown	1117	3381
BSA functional	s	138	960
	slowdown	2300	6000
BSA architectural	s	406	3344
	slowdown	6767	20900

Table 1: simulation speed.

The functional simulator is approximately 50 times slower than the comparable SimpleScalar functional simulator. This can be explained by a number of design decisions we have made: (i) the input is text-based (free format), which requires the input program to be parsed, (ii) the object-oriented design is responsible for a slowdown, (iii) the flexibility that was requested causes a major slowdown because anything should be dynamically modifiable, nothing could be optimized by the compiler (such as the number of registers, structure of a block, and so on), and (iv) last but not least, the BSA-architecture is more complicated than the SimpleScalar architecture, and hence, requires more resources to simulate, e.g., BSA-blocks puts instructions in different sections, according to their type (memory, floating point, one-cycle and multiple cycle arithmetic & logic); this means the functional BSA simulator has to retrieve the original program order, the SimpleScalar simulator does not.

The architectural simulator is only 6 times slower. Besides the causes just mentioned, this is in addition due to the extra detail of simulation (e.g. forwarding is explicitly modeled).

Table 2 shows the slowdown caused by the use of a reference execution to verify the operation of the simulator.

program	no reference (s)	reference (s)	slowdown
yacc	126	159	26%
lex	960	1265	32%
compress	347	466	34%
li	1242	1553	25%
ijpeg	803	1035	29%
average			29%

Table 2: functional simulation time

A slowdown (29% on average) is observed, which is acceptable, given the huge debugging benefits that come with it. The reason that the slowdown is limited is that the functional SimpleScalar simulator is about 50 times faster than the BSA-simulator, and hence, the additional resource consumption (2%) is hardly noticeable. The major source (remaining 27%) of slowdown is the communication overhead between the two simulators, and the verification itself.

The architectural simulator uses the functional simulation as a reference. The relative slowdown is now 37% as shown (See Table 3).

program	no reference (s)	reference (s)	slowdown
yacc	406	551	36%
lex	3344	4443	33%
compress	1009	1415	40%
li	3734	5249	41%
ijpeg	2568	3507	37%
average			37%

Table 3: simulation time for detailed, architectural simulation

Since the functional simulator is 3 times faster than the out-of-order one (Table 1), the simulation overhead is now the dominating factor. The communication and verification overhead can now be neglected.

5 Related work

Many different approaches are used to construct a simulation platform for computer architecture, but we have not found an approach that is similar to ours.

5.1 Trace-based simulations

Many simulators use an execution trace as input. Such a simulator is easy to develop: no functional component is needed [7] and hence, a trace-based execution is always 100% deterministic. This type of simulation is often sufficient to investigate the memory subsystem [5,8],

where the only information that is needed are data access streams which can be extracted from a trace.

An important disadvantage of trace-based simulation is the lack of information concerning misspeculations: a trace only mentions the committed instructions, whereas in an aggressive out-of-order processor core many instructions are in fact the result of misspeculation: a 4-way issue processor is reported to have an instruction overhead of 16% to 105% [6][1]. This information is not included in an execution trace.

5.2 Direct execution

Another technique is the translation of the code for the target architecture (i.e. the architecture being simulated) into a native program for the host (i.e. the machine where the simulation is executed). The resulting program emulates the target architecture. The code is instrumented to extract simulation results. This allows very fast simulation: the simulation code executes only seven to ten times slower than the native version [3]. However, if the level of detail increases, this speed advantage disappears.

This technique makes all architectural[2] information available to the instrumentation code, but not the detailed micro-architectural state. Consequently, the problems concerning misspeculations are inherited from trace-based simulation. Second, the simulator is bound to a particular host platform. Third, a textual interface, offering many advantages, is not possible.

This is however an interesting technique if one is only interested in functional simulation (to check compiler algorithms). It is often used for profiling purposes.

5.3 Interpreter

This type of simulation uses an executable as input (or an equivalent textual representation, as in our case). All necessary information is obtained by interpreting each instruction and maintaining the complete architectural state. This approach is far more difficult than using a trace: one has to implement a functional component and handle system calls. Additionally, system calls interact with the host environment. This weakens the reproducibility[3]. The main advantage is that one can construct a very accurate simulator, e.g. it is possible to explicitly follow a wrong branch prediction, so this is the best basis to construct a detailed architectural simulator, as done in [1].

In our case, trace-drive simulation was not usable for our first functional simulator platform, because, of course, there is no tool that can produce a trace for our Block Structured Architecture. An interpretative functional simulator was constructed, because this way we aren't bound to a particular host platform (direct execution), and because this is the best starting point for a detailed architectural simulator.

6 Future work

We have been using the two simulation platforms for more than one year now. They have become quite stable, and many experiments have been carried out with them (e.g., new branch prediction schemes, value prediction, caches have been added, gathering statistical information about executions, etc.). After the final iteration of the BSA-architecture, we plan to replace the existing text-based compiler/simulator interface by a binary interface. This will be done by removing the source code front-end from the simulators, and using this front-end as front-end for an assembler program. The simulator will get a new binary front-end.

Execution time is a weak point for all detailed simulators. Parallel machines can be an attractive means to alleviate this problem. Partitioning a simulation algorithm requires that the task can be divided into relatively independent subtasks (communication overhead); this technique is therefore mainly used in the simulation of multiprocessor architectures, where the partitioning is trivial [3,8].

A Block Structured Architecture that executes multiple blocks in parallel maps naturally on a multiprocessor host platform. We will investigate the usefulness of parallel simulation for this type of target architecture.

7 Conclusion

In this paper, we have described the bootstrapping process of a simulation platform for an experimental computer architecture (BSA). We claim that the development of an architectural simulator is greatly facilitated by the existence of a functional simulator that can be used as reference. That is why we started this work with the development of a functional simulator.

We have also described five techniques that had a serious impact on the development speed of the compiler and simulator: the use of a functional compiler implemented by means of a post-processor for an existing compiler, the use of a reference execution, an ASCII interface between the compiler and the simulator, an object oriented design, and trace-based debugging facilities.

---

[1] This problem can be circumvented by also using the object file [7], but this also makes a functional component necessary.

[2] The information visible to the programmer, e.g. register values, but not the invisible information such as branch prediction, caches, etc.

[3] This can be solved using I/O traces.

The simulation platform has now been in use for more than one year, and has been quite stable since the very beginning.

Acknowledgements

Bart Goeman is supported by FWO project 3G003699. Koen De Bosschere is research associate with the Fund for Scientific Research – Flanders. Henk Neefs is supported by GOA project 12.0508.95.

References:

[1] Doug Burger, Todd Austin and Steve Bennett, Evaluating Future Microprocessors: the SimpleScalar Tool Set, *University of Wisconsin-Madison*, TR-1342, 1997.

[2] Henk Neefs, Koen De Bosschere and Jan Van Campenhout, "Issues in Compilation for Fixed-Length Block Structured Instruction Set Architectures," in *Workshop on Interaction between Compilers and Computer Architectures, San Antonio (Texas)*, February 1997.

[3] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S-K. Chen, and A. Polyak, Simulation/evaluation environment for a VLIW processor architecture, *IBM Journal of Research & Development : Performance analysis and its impact on design (PAID)*, Vol. 41, No.3, 1997, pp.287-302.

[4] Bart Goeman, Development of a Simulator for a Block Structured Architecture, *Master's thesis, University of Ghent*, 1998.

[5] Luis Barriga, Mats Brorsson and Rassul Ayani, A model for Parallel Simulation of Distributed Shared Memory, *Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, San Jose, CA, February 1996.

[6] Srilatha Manne, Arthur Klauser and Dirk Grunewald, Pipeline Gating: Speculation Control for Energy Reduction, *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 132-141.

[7] Henk Neefs, Koen De Bosschere and Jan Van Campenhout, A C++ Simulator modelling a modern data-flow scheduling Microprocessor, *Seminar on Parallel Computing*, Noordwijk aan Zee (NL), October 1996

[8] Xiaohan Qin and Jean-Loup Baer, A Comparative Study of Conservative and Optimistic Trace-driven Simulations, Proceedings of the 28th Annual Simulation Symposium, 1995, pp.42-50.