A Programming Language Based Analysis of Operand Forwarding

Lennart Beringer

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
Mayfield Road, Edinburgh EH3 9JZ, UK

Abstract. We outline a programming language based analysis of forwarding. Abstractions of processor behaviour are modelled as operational semantics for a language which captures the hardware resources for forwarding explicitly. Unsafe usage of the forwarding mechanism is eliminated by static semantics. These type systems may be linked to static program analysis frameworks but also characterise the instruction stream entering the datapath from other processor components.

1 Introduction

The forwarding (register-bypassing) of operands is a technique implemented in many modern microprocessors [7] [5]. The formal correctness of forwarding mechanisms such as Tomasulo's algorithm [16], and their interaction with other elements of processor architecture have been widely studied [1][14] [10]. These verification efforts follow the well-known approach of relating processor implementations to the instruction set architecture [4] [9], using model checking and theorem proving. In this paper, we present a more conceptual analysis of forwarding using an abstract model of computation comprising named operand queues, registers and functional units. We demonstrate that using programming language notation yields an analysis which separates structural and implementational aspects of forwarding. As a consequence, we may reason about constraints on the allocation of operand queues to operands which are imposed by functionality considerations without committing ourselves to a particular allocation algorithm.

Elements of a Programming Language Based Approach. Our approach is programming language based as it builds on the methodology of modern programming language design, in particular on the separation into static and dynamic semantics. We use

- syntax for reflecting architectural entities or the fine-grained structure of instructions. We present a language in which the forwarding resources are explicit: the names of operand queues appear in the syntax of instructions, much like those of registers.
- structural operational semantics (SOS, [13]) for defining processor behaviour. By giving a language *several* operational semantics one may compare processor behaviour at various levels of abstraction. In this paper, we outline a processor model for sequential execution (similar to the ISA), while referring the reader to [3] for models for distributed (out-of-order) execution and execution with finite operand queues.

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 270–276, 2003.

– static semantics for formally expressing properties of the instruction stream which cannot easily be captured syntactically. In particular we employ type systems based on linear logic to characterise properties of the allocation of operand queues.

As program properties (static semantics) and processor behaviour (dynamic semantics) are tied together by syntax, proof techniques which are guided by the syntactic structure may be used for reasoning about properties of instruction streams in a particular processor model. The allocation of operand queues is thus validated according to the slogan *well typed programs can't go wrong*: an instruction stream which has been accepted by the static semantics will not to experience specific runtime hazards. The main such technique is *structural induction*, where the proof of a property of a certain phrase relies on related properties of syntactically constituent phrases.

Static semantics also allows system properties to be related to compile-time analysis. A system designer may hence explore whether properties of the instruction stream can better be ensured by the hardware or the compiler. For example, the decision whether a value is forwarded is often made in the control unit. Based on the static semantics, we present an alternative analysis using dataflow equations. This allows us to study the amount of forwardable values in application programs, but also indicates the type of analysis a hardware implementation must perform in order to exploit these forwarding opportunities.

Related Work. To our knowledge, no structural analysis of forwarding has been published yet, despite the plethora of verification exercises of processors with register bypassing.

The application of flat term rewriting systems (TRS) for describing and relating processor models is advocated in [2]. This formalism captures the structure of the processor in a similar way as our approach, and aspects of our computational model may be seen as a refinement of [2]'s substitution-based communication of operands. However, the operational model is not complemented by static semantics, hence program and processor may not be treated in combination and no link to compile time analysis may be made. On the other hand, [8] reports how hardware implementations may be directly generated from the TRS descriptions. We have not attempted this task but are confident that one could develop a corresponding translation from SOS-based descriptions.

Mountjoy et. al. [11] present a SOS description of transport-triggered architectures where the structure of the semantics follows the structure of the architecture. A single dynamic semantics is given which models the (synchronous) execution of a family of move-instructions in two phases. The authors observe that the legality of code relies on the ability of the compiler to structure code in a way which avoids output-conflicts. While static semantics is mentioned as a means to enforce this property, no details are given in [11] and the topic was apparently not pursued any further.

This paper represents a brief summary of the author's PhD thesis [3]. The reader is referred to [3] for a more in-depth presentation which includes formal proofs, a description of the experimental results, and more motivating discussion.

2 Syntax and Operational Semantics

We consider a simplified model where a processor core consists of a number of typed functional units which are located in parallel to each other and are fed by instruction queues. Operands are communicated through registers or operand queues, and the syntax of our language treats both mechanisms identically. We have instructions like $\texttt{add}\ op_1\ op_2\ op_3$, $\texttt{dupl}^{fu}\ op_1\ op_2\ op_3$ and $\texttt{if}\ op_1\ n_1\ n_2$ where the op_i denote registers or operand queues, fu represents a functional unit and the n_i denote program labels.

Specific processor models are defined by giving dynamic semantics for the language. The sequential model of operation is defined by a relation $\mathsf{C} \xrightarrow{t} \mathsf{D}$ between configurations C and D and an instruction sequence t. Configurations consist of a register bank, a component describing the content of all operand queues (technically a map from operand queue identifiers to sequences of values), and a memory component. On arrival at a functional unit, an instruction awaits its operands in the queues or registers as indicated in its opcode. Whenever its functional unit becomes available, the instruction executes which involves consuming operands from (and sending results to) registers and operand queues. The relation $\mathsf{C} \xrightarrow{t} \mathsf{D}$ is defined along the syntactic structure. Rules for individual instructions employ micro-instructions for read and write access to operand queues, registers and memory. For example, executing the sequence $[1]\texttt{ldc}\,4\,\mathsf{q}_1\ [2]\texttt{dupl}^{\texttt{MEM}}\,\mathsf{q}_1\,\mathsf{q}_1\,\mathsf{q}_2\ [3]\texttt{add}\,\mathsf{q}_1\,\mathsf{q}_2\,\mathsf{q}_2$ in an initially empty state leads to q_2 containing the single value 8 and q_1 being empty.

The dynamic semantics may be used to inspect the execution of programs by unfolding the derivation tree for judgements $\mathsf{C} \xrightarrow{t} \mathsf{D}$. Properties of the dynamic semantics such as determinism may be proven using structural induction.

Alternative Dynamic Semantics. In addition to the sequential semantics, [3] defines a semantics for distributed (super-scalar) execution where instructions interleave. No assumptions are made regarding the delays inside functional units. We also consider operand queues of finite length. The relationships between these semantics correspond to the verification conditions in traditional processor verification, but may be proven by structural induction. For example, the distributed model subsumes in-order execution but admits additional interleavings, governed by the availability of operands in the operand queues.

3 Static Semantics

In general, each dynamic model of execution gives rise to specific run time hazards, i.e. conditions under which some programs do not execute correctly. Static semantics allows one to detect many classes of hazards syntactically.

For the sequential model of operation, the typical hazard occurs if an instruction fails to execute due to the lack of operands in the appropriate operand queues. This condition depends on the initial configuration, but also on contextual instructions. In the case of a loop, such a deadlock may only become manifest after a number of iterations.

The type system we present employs a fragment of linear logic [6]. Referring the reader to [3] for formal details, we consider types which are linear products over the set of operand queues and registers, where registers are modelled by exponentials "!". We thus abstract from the particular values of operands and from the order of items in each queue. Our type system contains one axiom for each instruction form. To each instruction we associate a pair of types which relate configurations prior to the execution to the shape after the instruction has been executed. Typed instructions are composed to instruction sequences using a cut rule. Each straight-line sequence of code is again associated a pair of pre- and post-types. At branch points, we require the net effect of each loop body on the number of elements in each queue to be neutral, similar to work by Stata-Abadi [15]. Operands expected by a loop body must be provided by earlier basic blocks and all operands created in the body must either be consumed immediately or be passed on to successor basic blocks.

The inference of a typing derivation proceeds by weakening *minimal* typings of basic blocks until a unification of types at basic block boundaries is obtained. Failure of unification indicates the presence of a loop where each iteration consumes more values from the operand queues than it produces, or vice versa.

The soundness of the type system guarantees that well-typed code will not get stuck due to insufficiently many operands. The proof of this result proceeds by structural induction: first single instructions are considered by proving the soundness of the axioms, then straight-line code is considered by proving the soundness of the cut rule, and finally full programs are considered using the rule for combining basic blocks. Thus, well-typed programs will either diverge or will successfully complete, irrespectively of the number of loop iterations. The size of intermediate configurations is statically bound.

Alternative Models of Execution. In [3] we generalise our analysis to the alternative dynamic semantics. The typical hazards for distributed execution are race conditions and functional non-determinism as the execution of each instruction is triggered purely by the presence of operands. In our approach, these hazards are seen as a joint property of program and processor. Instead of immediately introducing hardware mechanisms for synchronisation we employ static semantics to identify non-deterministic programs. We extend the type system to detect race conditions and consider various techniques for guaranteeing that the corresponding serialisation requirements are met. Indeed, many programs may be serialised without additional synchronisation hardware.

A particular advantage of our analysis is observed for the model with operand queues of finite length. Here, the characteristic error condition consists of a deadlock due to an operand queue overflow. Our analysis shows that the absence of deadlock is preserved for deterministic programs when the length restrictions are relaxed, while for other programs this is in general not the case.

4 Program Analysis

The third aspect of a programming language based approach consists of the ability to formally relate low-level properties to program analysis frameworks [12]. We present a dataflow analysis for a labelled intermediate language for detecting when an intermediate

value is used exactly once. These read-once values are candidates for forwarding, as their single usage corresponds to the deletion from the operand queue during a read access.

The analysis targets the dynamic number of uses of an intermediate variable: any two assignments must be separated by exactly one read-access and no values should be left over at the end of a program run. We generalise the dataflow equations for liveness [12] by using a four-element lattice $\mathcal{L}$ and say that a pair of functions $\mathsf{fwd}_{entry}, \mathsf{fwd}_{exit} : \mathbf{Lab_P} \to \mathbf{Var_P} \to \mathcal{L}$ is a solution if

$$\mathsf{fwd}_{exit}(\ell)(x) = \begin{cases} 0 & \text{if } \ell \in \mathbf{final}(\mathbf{P}) \\ \sqcup_{(\ell,\ell')\in\mathbf{flow}(\mathbf{P})} \mathsf{fwd}_{entry}(\ell')(x) & \text{otherwise} \end{cases} \tag{1}$$

$$\mathsf{fwd}_{entry}(\ell)(x) = \begin{cases} uses(\ell)(x) & \text{if } x \in kill(\ell) \\ uses(\ell)(x) \oplus \mathsf{fwd}_{exit}(\ell)(x) & \text{otherwise} \end{cases} \tag{2}$$

where $kill$ and $uses$ are again generalisations of the corresponding functions in the analysis of liveness. The forwardability information of a solution is contained in the component fwd_{exit}. In [3] we show that a value assigned to a variable x at a program point ℓ is read exactly once if $\mathsf{fwd}_{exit}(\ell)(x) = 1$ holds. Variables x for which $x \in kill(\ell)$ implies $\mathsf{fwd}_{exit}(\ell)(x) = 1$ for all ℓ may thus be deleted after any read access. Notice the similarity to the characterisation of useless variables by liveness analysis. The proof of this characterisation formally relates (1) and (2) to a dynamic semantics of the intermediate language.

Compilation Based on Dataflow Solutions. Based on dataflow analysis, a compiler may convert intermediate programs into assembly code. The allocation of operand queues to read-once variables differs from register allocation as the order of writing must coincide with the order of reading. In [3] we demonstrate how conflict graphs between read-once variables may be obtained similarly to conflict graphs for register allocation and prove the functional correctness of a translation which maps adjacent variables to different operand queues. We also show that the resulting code is well-typed and thus structurally correct with respect to the underlying hardware. The existence of weakenings for satisfying the typing condition at basic block boundaries is guaranteed by the dataflow solutions, and loops are of neutral net effect. Indeed, the typing judgements may be formally obtained from the dataflow solutions, eliminating the need for an assembly level type inference.

Experimental Results. The dataflow analysis was implemented for two conversions of JVM code into the intermediate language and exercised on the Linpack benchmark suite. We observed that nearly all usage of the operand stack may be translated into forwarding if an SSA-like conversion scheme is used. Furthermore, the number of allocated registers decreased by up to 50%, even if each operand queue may only be used for operands sent to a specific functional unit. More significant than these static measures are dynamic measures: our analysis shows that on average 65% of the (central) register read operations turn into (local) operand queue reads, while the corresponding number for write operations is 62%.

5 Discussion

We presented an analysis of forwarding based on dynamic and static semantics of a language with explicit forwarding. We demonstrated the ability of programming language technology to eliminate important classes of error conditions (deadlocks and race conditions) and to analyse the forwarding potential of programs. Interpreting our language as the compiler-visible definition of a processor leads to a verification approach which emphasises that overall system correctness depends as much on program properties as on the correctness of processor implementations. On the other hand, it may be undesirable to expose operand queues explicitly to the programmer. Under this perspective, our analysis demonstrates how a separation between functional and implementational aspects of forwarding may be achieved. Future work is needed to identify how the dataflow-based compilation may be related to hardware implementations. Although the technical results apply only to the specific model of computation considered in this paper, we thus argue that type systems and other syntax-directed formalisms provide a solid basis for structured reasoning about interactions between processor architecture and compilation.

Acknowledgements. The author is grateful to Colin Stirling and Ian Stark for supervising the work described in this paper and for suggesting numerous presentational improvements.

References

1. T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. In *Proceedings of the 12th International Conference on VLSI Design*. IEEE Computer Society, 1999.
2. Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro, Special Issue on Modeling and Validation of Microprocessors*, 1999.
3. L. Beringer. *Asynchronous Queue Machines with Explicit Forwarding*. PhD thesis, School of Informatics, University of Edinburgh, 2002.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *LNCS*. Springer, 1994.
5. M. J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones & Bartlett Publishing, 1995.
6. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 46, 1986.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
8. J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the Tenth International Conference on VLSI (VLSI'99)*, Lisbon, Portugal, 1999.
9. M. D. Aagard and B. Cook and N. A. Day and R. B. Jones. A Framework for Microprocessor Correctness Statements. In T. Margaria and T. Melham, editor, *Proceedings of the 11th Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *LNCS*. Springer, 2001.
10. K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *LNCS*. Springer, 1998.

11. J. Mountjoy, P. Hartel, and H. Corporaal. Modular Operational Semantic Specification of Transport Triggered Architectures. In *Proceedings of the 13th Conference on Computer Hardware Description Languages and Their Applications (CHDL'97)*. Chapman and Hall, London, 1997.
12. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
13. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, University of Aarhus, 1981.
14. J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editor, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *LNCS*. Springer, 1998.
15. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 1998.
16. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and development*, 11, 1967.