Hardware Support for Large Atomic Units in Dynamically Scheduled Machines

Stephen W. Melvin
Michael C. Shebanow
Yale N. Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Microarchitectures that implement conventional instruction set architectures are usually limited in that they are only able to execute a small number of microoperations concurrently. This limitation is due in part to the fact that the units of work that the hardware treats as indivisible are small. While this limitation is not important for microarchitectures with a low level of functionality, it can be significant if the goal is to build hardware that can support a large number of microoperations executing concurrently. In this paper we address the tradeoffs associated with the sizes of the various units of work that a processor considers indivisible, or atomic. We argue that by allowing larger units of work to be atomic, restrictions on concurrent operation are reduced and performance is increased. We outline the implementation of a front end for a dynamically scheduled processor with hardware support for large atomic units. We discuss tradeoffs in the design and show that with a modest investment in hardware, the run-time advantages of large atomic units can be realized without the need to alter the instruction set architecture.

1. Introduction

In this paper we discuss the notion of an indivisible unit of work, or an 'atomic unit.' When we refer to the 'size' of such a unit, we mean the the number of 'microoperations' that implement the atomic unit. A microoperation is a basic unit of work like a register-to-register move, a simple arithmetic operation, a load or a store.

We have identified three separate types of atomic units: *architectural atomic units, compiler atomic units.* and *execution atomic units.* An architectural atomic unit is a group of microoperations which is atomic with respect to interrupts and exceptions. The state of execution can only be preserved at an architectural atomic unit boundary. An architectural atomic unit is what is referred to as an 'instruction'. The specification of these units is important from the standpoint of interrupt latency and exception handling.

The compiler atomic units are the smallest indivisible units of work that the compiler generates. In most cases the compiler and architectural atomic units are exactly the same, but in principle the architectural atomic units could be larger. In that case, the compiler would group together a set of individually produced units of work and define them to be uninterruptible. The definitions of the architectural atomic units and the compiler atomic units together form what is generally called the instruction set architecture.

An important issue which relates to the format of the compiler atomic units is the notion of the degree of 'coupling' to the implementation. The compiler atomic units may be freely formated, in which case they are decoupled from the implementation, or alternatively they may be formated exactly into what we refer to as 'multinodewords' (which are the units of work that a machine can issue in one cycle). We discuss this issue further below.

The execution atomic units are the smallest groups of microoperations that the hardware issues at run-time as an indivisible unit. In sequential, non-pipelined microarchitectures, the execution atomic units are generally the individual microoperations. This is because the execution of each microoperation is dependent on the outcome of the previous microoperation (since a potential microbranch or microtrap exists). As pipelining is added and the number of function units is increased, increasing the size of the execution atomic units allows higher utilization and/or decreases the associated hardware cost by reducing the number of microoperation boundaries that are preserved.

In most implementations the execution atomic units are no larger than the architectural atomic units. This is because allowing the execution atomic units to increase beyond the size of the architectural atomic units is non-trivial and requires the implementation of some sort of backup hardware [1]. In the event of an exception in the middle of an execution atomic unit (but at the boundary of an architectural atomic unit), the backup hardware restores the state of a machine to the beginning of the execution atomic unit. Execution then proceeds one architectural atomic unit at a time until the exception is discovered. We define the operation of a machine implementing this backup hardware as having two modes: fast and slow. In fast mode, the machine executes the execution atomic units (which are larger than the architectural atomic units). In slow mode, the machine executes the architectural atomic units.

As an alternative, the architectural atomic units can be made equal to the execution atomic units. In either case, if the execution atomic units are made larger than compiler atomic units, the mechanism for packing compiler atomic units into execution atomic units becomes an important issue. This packing can either be done statically by the compiler or it can be done dynamically by the hardware.

If the compiler atomic units are closely coupled to the implementation (i.e. they are formated into multinodewords), as would be the case for a statically scheduled machine or for some dynamically scheduled machines [2], the compiler does the packing statically. If, however, the compiler atomic units are not closely coupled to the implementation, as would be the case if a buffer against changes in technology is desired or if an existing architecture is being implemented, the packing must be done dynamically. In this paper, we discuss a hardware mechanism called a 'fill unit' for dynamically creating execution atomic units from smaller compiler atomic units.

This paper contains four sections. Section two presents a general discussion of atomic unit sizes and the tradeoffs associated with them. Section three provides details on the implementation of a fill unit. Finally section four provides some concluding remarks.

2. Atomic Unit Size Issues

Many tradeoffs exist for choosing the sizes of the atomic units and it is important to identify which type of atomic unit is relevant for a particular tradeoff. Each of the three types of atomic units introduced above have distinct performance implications.

Larger architectural atomic units increase interrupt latency but potentially decrease the size of the architectural state of the machine. This is due to the fact that within an uninterruptible block of microoperations, temporary results need not be given explicit names, thus fewer general purpose registers are required. This makes context switching less expensive.

In addition, larger architectural atomic units simplify the design of high performance machines since there are fewer microoperation boundaries to preserve. This can be significant for microarchitectures with a high degree of concurrency. The third advantage of large architectural atomic units is that they make it easier to increase the size of execution atomic units as mentioned above (no fast mode slow mode scheme is required).

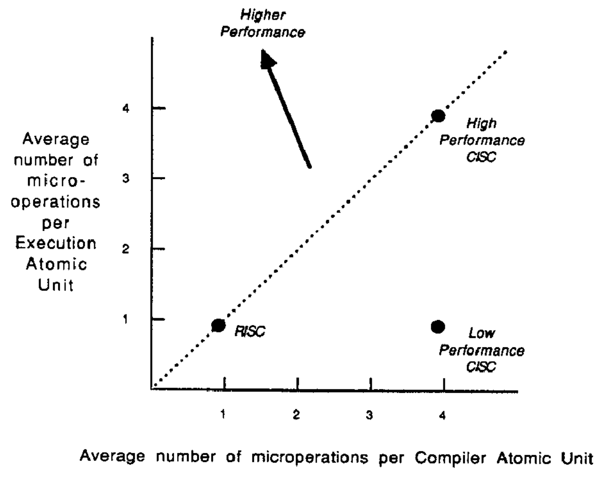
However, the architectural atomic units are secondary to performance. The sizes of the compiler atomic units and execution atomic units are more critical. Small compiler atomic units increase compile-time effort but allow more compile-time optimization by allowing a finer level of control over the work to be done. Larger execution atomic units allow a higher degree of parallelism to be taken advantage of more efficiently. By issuing a larger unit of work as a single unit, more concurrency can be achieved at a lower cost.

In a RISC machine, small compiler atomic units are implemented in order to exploit their advantages. Since most processors have architectural atomic units which are the same as the compiler atomic units, and since increasing the execution atomic units beyond the size of the architectural atomic units is complex, the result has been processors with small execution atomic units. Thus, while these machines can benefit from the performance advantages of small compiler atomic units, they are limited by the performance disadvantages of small execution atomic units.

Figure 1 illustrates this point. Consider a hypothetical RISC machine in which the compiler specifies individual microoperations and two implementations of a hypothetical CISC machine in which the average number of microoperations per instruction is four. In the high performance CISC case, we assume that the architectural atomic units are the same as the compiler atomic units and this limits the size of the execution atomic units. As one moves further up and to the left on this graph (meaning larger execution atomic units and smaller compiler atomic units), performance increases. Furthermore, we believe that for high performance processors, the size of the execution atomic units is more critical than the size of the compiler atomic units.

In summary, for higher performance it is desirable to have large execution atomic units and small compiler atomic units. In order to make the large execution atomic units less expensive to implement, large architectural atomic units are also desired. However, a designer implementing a given instruction set architecture cannot choose the

Figure 1
Atomic Unit Size Diagram



composition of the compiler atomic units or the architectural atomic units. The goal then is to implement execution atomic units which are as large as possible.

In the next section we describe a mechanism we call a 'fill unit' which allows large execution atomic units to be implemented in a dynamically scheduled machine given small compiler atomic units decoupled from the machine implementation.

3. Implementation Details

3.1. Instruction Unit Overview

Figure 2 shows an overview of a hypothetical instruction fetch and decode unit. Instructions are fetched from main memory or from an instruction cache and stored in a pre-fetch buffer. Depending on the format of the compiler atomic units, microoperations may be directly accessible or a microoperation generation stage may be necessary. In either case, a sequence of microoperations is passed on to the fill unit.

The fill unit packs microoperations into execution atomic units which are then passed on to the decoded instruction cache. In the frequently occurring case, execution atomic units are fetched directly from the decoded instruction cache and merged into the execution unit, after having register aliases resolved by the register alias table. The branch predictor predicts branches and sequences the execution atomic units.

A checkpoint retirement system is present to retire execution atomic units as they become fully executed. When an exception arises within a particular execution atomic unit, the machine is backed up to the previous boundary. Then, if the architectural atomic units are smaller than the execution atomic units, a bypass mode is entered (slow mode) where microoperations bypass the fill unit and the decoded instruction cache. Otherwise, the machine proceeds with the appropriate course of action for the exception which was detected without the need to reissue any microoperations. We will now consider the fill unit in more detail.

Figure 2
Fetch and Decode Unit

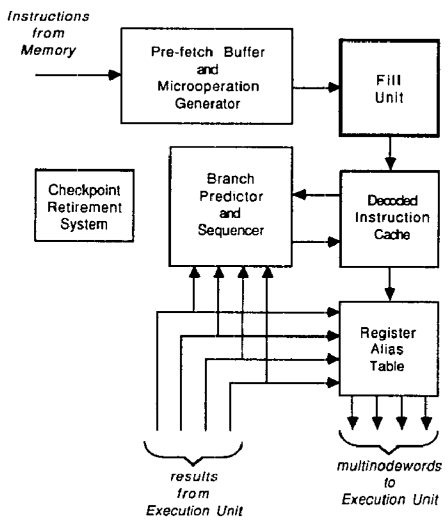
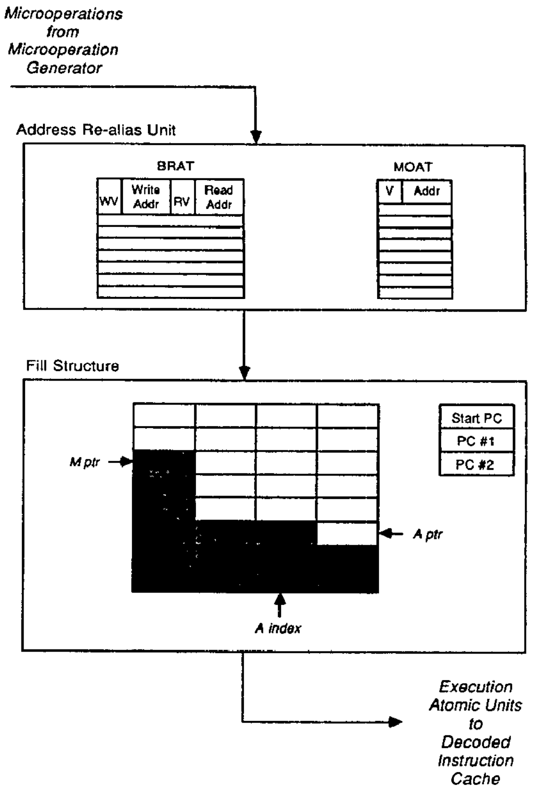

Instructions from Memory
Pre-fetch Buffer and Microoperation Generator
Fill Unit
Checkpoint Retirement System
Branch Predictor and Sequencer
Decoded Instruction Cache
Register Alias Table
results from Execution Unit
multinodewords to Execution Unit

Figure 3
Fill Unit



3.2. Fill Unit

The fill unit accepts microoperations one at a time and forms execution atomic units. In so doing, it allows the microoperation generator (or the compiler) to specify microoperations without having to know their final locations within a multinodeword or within an execution atomic unit.

A diagram of a hypothetical fill unit is shown in figure 3. We show in this example a fill unit which supports a multinodeword with four microoperations, one memory and three ALU. The address re-alias unit changes the addresses of microoperations to support the execution atomic unit format. The fill structure consists of an execution atomic unit which is under construction. This structure has the format of an entry in the decoded instruction cache.

Each operand in each microoperation is either a constant, a general purpose register reference, or a reference to another microoperation. If an operand is a constant, no work is required, the constant is simply copied into the fill structure. If an operand is a register reference, the re-alias unit uses the block register alias table (BRAT) to determine if a different address should be inserted. Finally, if an operand refers to another microoperation, it is assumed to be specified relative to the first microoperation in the instruction. In this case the re-alias unit will use the microoperation alias table (MOAT) to translate the reference. These re-aliases ensure that dependencies within a multinodeword and within an execution atomic unit are satisfied when the unit is merged into the data path.

Each entry in the BRAT corresponds to a general purpose register. There are four fields in each entry: the write address field, the write valid bit (WV), the read address field and the read valid bit (RV). The WV bit indicates validity of the write address field and the RV bit indicates validity of the read address field. Whenever a new execution atomic unit is created, all BRAT WV and RV bits are reset to signify that no writes have taken place within that unit.

If an operand reads the value in a register, the BRAT RV bit for that register is consulted. If the bit is reset, the register reference is left unchanged (at merge time, the RAT will translate the reference). If the RV bit is set, however, the register reference is changed into a relative microoperation reference (from the read address field). This situation indicates that a previously filled microoperation has written to the register. If an operand writes to a register, the WV bit is set and the write address field is updated with the fill structure address of that microoperation.

When the last microoperation for an instruction is sent, the BRAT is signaled to copy the WV and write address fields into the RV and read address fields for each entry. This mechanism allows *the microoperation generator (or the compiler) to generate reads and* writes to registers in any order within an instruction, while preserving inter-instruction, intra-execution-atomic-unit semantics.

One last detail is how an execution atomic unit is *finalized*, which occurs in two situations. The first is when a change in the flow of control is detected. Information regarding the two alternate PCs (assuming two way branches) and which microoperation will be generating the test condition are stored in the fill structure. The contents of the fill structure are then transferred to the decoded instruction cache where they will be stored for repeated use and merged into the main execution unit.

The second situation in which an execution atomic unit is finalized is when there are not enough empty microoperation slots for the next instruction to be generated. The microoperation generator signals to the fill unit at the start of each instruction how many free slots are required for each of the microoperation types (note that this can be a worst case estimate). If the fill structure has less than this number of free slots, it will finalize the unit, store control information indicating an unconditional branch and start a new unit. We assume here that an execution atomic unit is large enough to completely contain even the largest instruction.

4. Conclusions

In a microarchitecture with multiple pipelined function units, the goal is to keep as many pipeline slots full as possible. A dynamically scheduled processor attempts to do this by merging multiple microoperations per cycle and scheduling them out-of-order whenever their operands are ready. A problem that arises is that of low utilization of microoperations within multinodewords. The solution is to better pack the microoperations into the multinodewords and this can be done statically or dynamically.

In cases where the compiler atomic units are formated to the multinodeword structure, the compiler can pack the microoperations into the execution atomic units statically. This scheme cannot take advantage of run-time information (such as branch prediction information or dynamic operand values) and it locks the architecture to the implementation.

Alternatively, in order to support compiler atomic units which are not formated to the multinodeword structure and/or to decouple the architecture from the implementation, a fill unit is useful. The additional overhead of run-time filling is incurred but with a suitably large decoded instruction cache, this can be minimized. Thus, the performance advantages of large execution atomic units can be exploited without the need to lock the architecture into a specific multinodeword format.

Acknowledgement

Our work in microarchitecture is part of a larger architectural research effort at Berkeley, the Aquarius Project. We acknowledge our colleagues in the Aquarius group for the stimulating environment they provide. We also acknowledge that part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

REFERENCES

[1] Wen-mei W. Hwu and Yale N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines," *IEEE Transactions on Computers*, December, 1987.

[2] Wen-mei W. Hwu and Yale N. Patt, "HPSm, A High Performance Restricted Data Flow Architecuture Having Minimal Functionality," *Proceedings, 13th Annual International Symposium on Computer Architecture*, Tokyo, June, 1986.

[3] Yale N. Patt, Michael C. Shebanow, Wen-mei Hwu and Stephen W. Melvin, "A C Compiler for HPS I, A Highly Parallel Execution Engine," *Proceedings, 19th Hawaii International Conference on System Sciences*, Honolulu, HI, January 1986.

[4] Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, Michael C. Shebanow, Chien Chen and Jiajuin Wei, "Run-time Generation of HPS Microinstructions From a VAX Instruction Stream," *Proceedings, 19th Annual Workshop on Microprogramming*, October 15-17, 1986, New York, NY.