Block Based Execution and Task Level Parallelism

Richard H. Littin, J. A. David McWha,
Murray W. Pearson and John G. Cleary

Department of Computer Science,
University of Waikato,
Hamilton, New Zealand
{rhl, jadm, mpearson, jcleary}@cs.waikato.ac.nz

Abstract. A fixed-length block-based instruction set architecture (ISA) based on dataflow techniques is described. This ISA is the compared and contrasted to those of more conventional architectures and other developmental architectures. A control mechanism to allow blocks to be executed in parallel, so that the original control flow is maintained, is presented. A brief description of the hardware required to realize this mechanism is given.

1 Introduction

Modern computer architectures are approaching the limit of easily available performance gains through advances in manufacturing technologies. Increasingly they must try to improve performance by extracting parallelism. One family of techniques used to do this is out-of-order instruction execution. Through the use of reservation stations [Tomasulo, 1967] and reorder buffers, instructions can be executed in an order other than that of the original code. This can make better use of the computational resources available. However, steps must be taken to ensure that the results of the execution remain the same as a sequential execution, placing restrictions on the reordering which may be done.

Many studies have been performed [Wall, 1991; Lam and Wilson, 1992] to evaluate the potential parallelism available in sequential code. In most of these studies the major emphasis has been on the effects that control flow has on parallelism. They have looked at the performance issues associated with different forms of branch prediction and speculation.

Microprocessors require a window of instructions to keep the pipeline full and the functional units busy. Increasing the size of this window increases the probability that the functional units can be fully utilized. Branch prediction mechanisms are one means that can be used to increase the window size. However, they are not perfect, so after only a few branches the probability that the correct branch is being followed is too low to be useful [Perleberg and Smith, 1993].

An alternative is to extract coarse-grained parallelism by executing multiple blocks of instructions in parallel. This requires some way of ensuring that data dependencies between blocks are preserved. We examine a particular architecture,

the WarpEngine [Cleary et al., 1995]. The facilities for passing data between blocks are described in section 4. We also examine ways of mapping control flow onto blocks so that coarse grained parallelism can be extracted. The advantages of using explicit blocks for optimistic execution are described in section 5.

2 Architecture Evolution

Contemporary production architectures, such as the Pentium Pro [INTEL, 1995], can reorder instructions within a fixed size instruction pool. Although instructions can be executed out-of-order (consistent with dataflow constraints) they must be retired from the system in-order. The instruction window is of fixed size and its movement to introduce fresh instructions is constrained by the completion and subsequent in-order retirement of earlier instructions. In order to increase the parallelism extracted (and hence performance) the instruction window can be made larger. However, in the case of the Pentium Pro the hardware complexity of the reorder logic is $\Theta(N^2)$, where N is the size of the window.

In the Wisconsin Multiscalar Machine [Sohi et al., 1995] code is split up into blocks, with several blocks executing in parallel on essentially independent sequential CPUs. These blocks, known as *tasks*, may be as large as a whole program or as as small as an individual instruction. Each task is treated as an independent program. Branches to locations outside the block determine which block should follow the currently executing one. Between the blocks there may be both control and data dependencies which are maintained through *task squashing* and an Address Resolution Buffer (ARB), respectively. Potential parallelism is increased, but since there is now some speculative execution taking place, information needs to be stored so that incorrect computation can be rectified.

With variable sized blocks, the amount of information to be *state saved* is unknown. By fixing the size of a block the problem of efficient state saving becomes tractable in hardware. This raises an interesting issue about the structure of individual blocks, that is, whether they should be fixed or variable length.

Another advantage of fixed sized instruction blocks is that the *reorder buffer* can be tuned to a fixed number instructions. Neefs [Neefs and Van Campenhout, 1996] describes a fixed-length block structured instruction set architecture that removes the register renaming and dispatch stages from an out-of-order CPU pipeline. This reduces the length of the pipeline with register renaming being performed by the compiler. Instructions are fired in dataflow order, extracting the highest amount of parallelism possible within a block. A more detailed description of the advantages and disadvantages of block structured architectures can be found in [Neefs, 1996].

In the WarpEngine instruction set [Cleary, 1995] this concept is taken one step further by removing restrictions on the number of each type of instruction in a block. This allows for better filling of blocks. In the WarpEngine multiple blocks are executed in parallel, speculatively, providing parallelism at the block level, as well as the instruction level. Synchronization and dependency correctness are implemented using the Time Warp algorithm [Pearson et al., 1997].

Neefs' ISA and the WarpEngine are both examples of fixed sized block structured ISAs. We will follow Neefs [Neefs and Van Campenhout, 1996] in referring to these as fixed length Block Structured Architectures (BSA). In a BSA registers have unique names within a block, which are determined statically at compile time, taking register renaming away from the hardware.

3 Speculation with BSA

The main thrust of this paper is to show the advantages of BSAs for parallel execution. A sequential processor would execute blocks one after another. The control and data dependencies for such a machine are shown in figure 1 a).

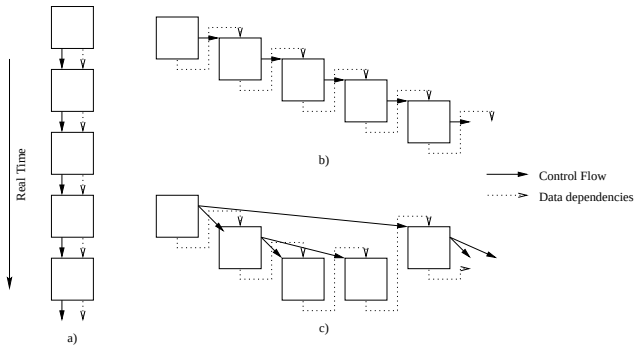

Fig. 1. Block control and data flow for a) a sequential machine, b) the Multiscalar Machine, and c) the WarpEngine.

If blocks are invoked speculatively then computation within a block can be overlapped with that of other blocks. Figure 1 b) shows how a speculative control mechanism, such as the one in the Multiscalar Machine, controls block invocation. Each block is started (and retired) in order but execution may overlap with the previous block. Speculation is used when determining which block to next fire in the chain. If an incorrect decision is made then the wrongly executed block and its successors are squashed and the correct block is then invoked. There is still a linear control sequence amongst the blocks.

In this model communication between blocks is achieved via standard register and memory accesses. If a location is read speculatively and the value returned is wrong (determined by a write to that location) then the affected instruction and any of its descendents need to be re-executed with the new correct value. This requires some form of state saving. In the Multiscalar Machine state saving is achieved by retaining block invocation information. Any data violations are satisfied by restarting the block of the affected instruction from its beginning (the register and memory reads then get the latest versions of values).

With this form of state saving some computation is unnecessarily undone as instructions that are prior to, or independent of, an affected instruction are also

re-executed. To get better performance finer grained state saving is required. This is achievable in a BSA since registers in a block are all single-assignment, therefore any changes to the data in a register occurs because the previous value was incorrectly (speculatively) calculated. The WarpEngine uses this level of state saving and the Time Warp mechanism to maintain the correct ordering amongst the states [Pearson et al., 1997].

With this finer grained state saving and the use of the Time Warp mechanism in hardware, the WarpEngine can use a control mechanism similar to that shown in figure 1 c). Here blocks are scheduled for execution in a tree structured manner. A tree can be generated through knowledge of *code convergence*. If an earlier block has been incorrectly invoked then only blocks that are direct descendents of that block need to be squashed. Blocks in other parallel sub-trees can keep processing.

4 Block control

This section describes how a tree is generated to control the optimistic parallel execution of sequential code. The program code blocks are mapped onto a control flow tree. The program's virtual ordering is maintained by traversing the tree top-down and left-to-right (a pre-order traversal). But the nodes in the tree are executed as they are created—top-down.

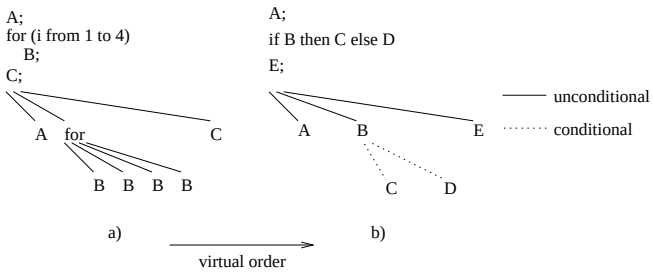

Fig. 2. Control trees for simple code structures; a) a for loop, and b) a conditional statement surrounded by unconditionally executed statements.

Figure 2 shows how the basic blocks in code segments are transformed into tree control structures. The dotted lines indicate conditional node creation, where the dependent node is not evaluated until the calculation of the conditional test has been completed. Figure 2 b) shows how control flow convergence can be exploited. Basic blocks A, B and E are guaranteed to execute, whereas C and D are dependent on the result of B. If A, B and E are largely independent, starting them early increases the parallelism.

Each basic block of the program is mapped onto a node in the control tree. Data flows in two ways: either through the memory system (when the sources and destinations cannot be determined statically); or by direct transfer between parent and child blocks or near relatives (this corresponds to transfers that would be done via registers in more conventional architectures).

5 Architecture

Any architecture that is going to implement this tree structured execution model needs to deal with causal violations that arise from data dependencies by either preventing them from occurring or detecting and correcting any that do occur. The WarpEngine architecture introduced here uses a combination of these. For dependencies within a block or its immediate descendants (children), using a dataflow mechanism prevents causal violations. For any other data dependencies that may exist between blocks a timestamped memory capable of detecting and correcting causal violations is used.

5.1 Optimism

The WarpEngine supports a form of speculation on memory reads which we refer to as *optimism*. Instructions that are dependent upon other earlier instructions, including reads, wait until the results of the instructions are available before executing. In the case of reads the results may be returned early and speculatively. For example the read may locate a value of the correct location but be uncertain as to whether it is the correct (most recently written) value. The value will be returned anyway, allowing the dependent instructions to proceed. If the value was in fact correct then all is well. If the value was incorrect then the read is re-executed and the dependent instructions (and any instructions dependent on them) are also re-executed.

5.2 Timestamped Memory

As memory read and write operations can occur out-of-order some mechanism is required to synchronize the reads and writes to memory to make sure that no causal violations occur. The WarpEngine uses a timestamped memory system that is based on Time Warp principles [Jefferson, 1985]. The memory system receives from the CPU read and write request messages and sends back reply messages in response to the reads. Each read and write request is accompanied by a timestamp. These timestamps impose a single linear temporal order relating all the reads and writes. The value returned for a read is the value from the last write prior to the read (in timestamp order) for that address. Note that we are assuming an optimistic system here, so that writes may be generated in a different order from their timestamps. This means a read which is satisfied by an earlier write may need to be re-satisfied if a write with a timestamp between the read and the earlier write arrives later in real time. Further details of the timestamped memory system can be found in [Cleary et al., 1995].

5.3 Frames

In the WarpEngine instructions are organized into blocks that roughly correspond to a basic block in classical CPUs. The only instructions that can be conditionally executed within a block are the instructions associated with scheduling the

block's children. A block has a maximum number of fixed-width instructions (in the current incarnation of the instruction set it is 16 instructions). Instructions specify the destinations of their results rather than the operand sources. Figure 3 illustrates how this mechanism works.

In memory each instruction is represented by two words. The first, the I-word contains the op-code and related information. The second, the C-word, contains a single literal or constant value that can be used as one of the instructions operands. An instruction goes through a number of stages in its execution. The block is *initiated* and is allocated a *frame*. A frame resides in the CPU and consists of 16 slots, one for each instruction. A slot contains three fields: an op-code field and two registers.

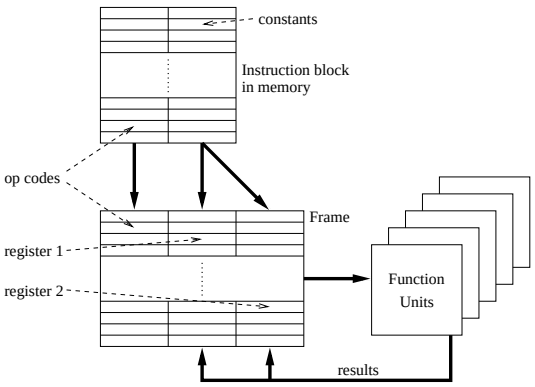

Fig. 3. Flow of Instructions in CPU.

The op-code is loaded directly from the instruction's I-word. Registers within a slot can be loaded either from the C-word of an instruction or as a result of another instruction. Once both registers in a slot have been filled with valid values then the instruction can be transferred to a functional unit for execution. The functional unit performs its computation returning up to four results back to the appropriate registers in the frame.

In addition to extracting maximum parallelism within a block, the frame structure provides an ideal state saving mechanism. All resources necessary for state saving the block are allocated as part of scheduling the block for execution. It offers a hierarchical form of register renaming without the complex resources that are required to implement it dynamically.

6 Experimental Framework

This section describes the methodology used to analyze the potential parallelism that can be obtained from a WarpEngine based architecture. Instruction counts are obtained both from a WarpEngine simulator and from SPIM, a MIPS instruction set simulator [Larus, 1993].

6.1 Programs

The programs which we have simulated span the types of operations that are performed in many programs including sorting, dynamic structure manipulation, matrix/array operations and recursion. The algorithms are simple in concept, but vary in the relative amounts of data and control dependence.

Two tree insertion routines are examined. A naive binary tree insertion (bin) and the AVL algorithm (avl). Sorting is covered by HeapSort (heap) and two versions of QuickSort (qu1 & qu2) which differ in the manner in which they select a pivot point. Matrix multiplication (mat), Gauss-Jordan elimination (gj) and transitive closure (trans) are all examples of array based manipulations.

6.2 WarpEngine Simulator

Each algorithm is hand coded from C to an assembly level language. Evaluation of the potential parallelism within an algorithm is achieved by taking the machine executable and running it on a functional level simulator of the WarpEngine. This simulator performs a pre-order traversal of the control tree executing the code at each node. The program is run in its virtual order and information about the real time that events occur is retained and used to determine the parallel execution time. The reported measure in each case is the potential parallelism–total time for the execution of all instructions divided by the time the last instruction completes. The simulator assumes that an infinite number of CPUs are available and that there are no constraints due to finite bandwidth to memory or cache systems.

7 Results

In this section we analyze the instruction counts for a WarpEngine based BSA and compare it to the of a MIPS instruction set. We examine the block utilization (number of useful instructions in the block) and also the performance gains that are possible when using a BSA.

Table 1 gives compiled and executed instruction counts for each of the algorithms for the MIPS and WarpEngine instruction sets. The *norm.* and *opt.* fields in the MIPS section correspond to the gcc compiler optimization levels 'O0' and 'O3' respectively. The dynamic instruction counts for the WarpEngine are comparable to those of the MIPS ISA, with the majority falling between the normal and optimised MIPS instruction counts. Extra instructions are necessary because there are more control instructions in a BSA, especially for straight line code.

The algorithm/problem size combinations in Table 1 show block instruction utilization of between 48.1% and 63.0% (mean 53.3%, standard deviation 5.1%) for the static compiled code, and ranges between 34.6% and 67.9% (mean 47.8%, standard deviation 9.9%) for the dynamic executed code.

		MIPS				*WarpEngine*			
	problem	*static*		*dynamic*		*static*		*dynamic*	
code	*size*	*norm.*	*opt.*	*norm.*	*opt.*	*insts.*	*util.*	*insts.*	*util.*
avl	2000	353	373	1023887	456406	462	48.1%	1497710	43.3%
bin	2000	97	89	896869	420156	134	55.8%	471070	45.8%
gj	25	701	303	1435987	552146	492	50.4%	911488	50.9%
heap	2000	363	187	3306575	1459528	185	55.1%	1354999	53.1%
mat	50	100	79	5450731	1402770	107	55.8%	3578560	46.6%
qu1	2000	155	67	995247	320967	143	49.6%	706433	40.2%
qu2	2000	152	79	892772	407571	248	48.4%	1090203	34.6%
trans	30	198	79	2019334	345177	121	63.0%	1457056	67.9%

Table 1. MIPS and WarpEngine instruction counts and block utilization

code	*problem size*				
	100	200	500	1000	
avl	19.4	29.7	49.7	75.6	
bin	27.2	39.3	59.9	75.8	
heap	12.4	13.2	16.0	17.5	
qu1	22.2	35.6	53.3	80.9	
qu2	4.1	4.4	4.3	6.2	
	5	10	15	20	25
gj	39.9	101.6	179.6	340.1	393.3
trans	82.9	412.4	1133.0	2213.7	3808.8
	5	10	20	30	40
mat	56.8	267.4	1417.0	3784.2	7440.3

Table 2. Potential parallelism for various problem sizes.

When executing just one block at a time on the WarpEngine simulator the average parallelism extracted was 1.62 (standard deviation, 0.15) with all instructions set to 1 cycle. When instruction timings were set to more typical values the average parallelism extracted rose to 2.24 (standard deviation, 0.29). These numbers confirm Neefs' [Neefs, 1996] assertion that there is less instruction level parallelism in a single threaded BSA than in a conventional ISA.

Table 2 shows potential parallelism for each algorithm/problem size combination using typical instruction cycle times. In all but qu2 the amount of parallelism obtainable increases with the problem size. This shows that there is significant parallelism available in sequential code which can be extracted by optimistic BSAs.

Figure 4 shows the potential parallelism that is available in the sorting algorithms tested. The difference between the parallelism curves of the two QuickSort routines is significant. Although the same underlying algorithmic concept is used the manner in which the pivot point is selected is critical. In this case the choice of algorithm has affected the data dependencies. With the correct

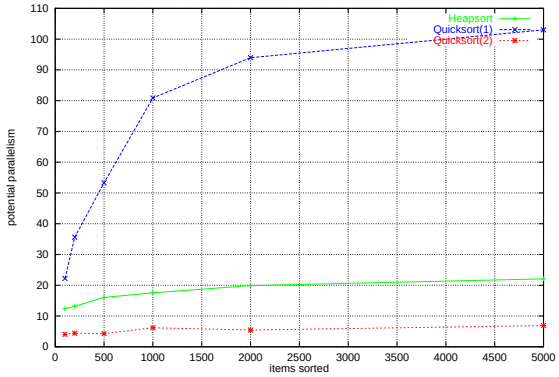

Fig. 4. Sorting, potential parallelism vs problem size.

choice of algorithm the BSA is able to support significant potential parallelism. Final transformation of this potential into execution speed up requires careful implementation of the underlying execution mechanisms.

8 Conclusions

We have demonstrated how Block Structured Architectures can be used in a CPU that is designed to extract parallelism at both the instruction and task level. At the instruction level a simple dataflow mechanism was used to communicate between instructions. The result is that parallelism and instruction re-ordering are possible without the need for reservation stations or dynamic checks on whether instructions can be re-ordered. Thus the benefits of modern high performance CPUs can be extracted with significantly reduced complexity.

At the level of task parallelism the WarpEngine uses a system of timestamping memory accesses and optimistic control to allow tasks to be executed in parallel even when there are potential data dependencies. The use of blocks enables a rich set of communication mechanisms for data. When no static information is available about the usage of a variable then communication is via relatively expensive time-stamped memory. When static information is available then data can be directed to a particular instruction within a block. This can be done directly to the children of the sending block or indirectly to more remote blocks. That is, low overhead mechanisms can be used when static information is available and higher overhead mechanism are needed only in the fully dynamic case.

Blocks also provide a convenient unit for controlling the task parallelism. For example a block of about 16 instructions fits well with what is needed for tree structured control within loops. As well many overheads can be dealt with at the block level, thus amortizing them over more instructions. These include fetching the instructions, providing resources for state saving and rollback, and retirement.

One problem has been identified from our study. That is the increased amount of code that needs to be fetched and stored. This arises from a number of sources. First, there is a small increase in the number of control instructions because blocks tend to break up the control of sequential code. Second, it is not possible to always fill all the instructions in a block. The measured efficiencies are around 50%. This problem can be addressed either through improving the code generation (the test examples were hand coded assembler) or through altering the ISA to allow a more compact representation of code.

References

Cleary, John G. (1995). WarpEngine instruction set. Internet Web Page. URL http://www.cs.waikato.ac.nz/timewarp/wengine/instset/index.html.

Cleary, John G., Pearson, Murray W., and Kinawi, Husam (1995). The architecture of an optimistic CPU: The WarpEngine. In *Proceedings of HICSS*, volume 1, pages 163–172, Hawaii.

INTEL (1995). PENTIUM PRO processor at 150 mhz. INTEL Corporation Datasheets. Order Number: 242769-001.

Jefferson, David (1985). Virtual time. *Transactions on Programming Languages and Systems*, 7(3):404–425.

Lam, M. S. and Wilson, R. P. (1992). Limits of control flow on parallelism. In *19th Annual International Symposium On Computer Architecture*, pages 46–57, New York, N.Y. ACM.

Larus, James R. (1993). *SPIM S20: A MIPS R2000 Simulator*. Computer Sciences Department, University of Wisconsin-Madison.

Neefs, Henk (1996). A preliminary study of a fixed-length block-structured instruction set architecture. Technical Report 96-07, Eletronics and Information Systems Department, University of Gent, Belgium.

Neefs, Henk and Van Campenhout, Jan (1996). A microarchitecture for a fixed length block structured instruction set architecture. In *Eighth IASTED International Conference on Parallel and Distributed Computing and Systems*.

Pearson, Murray W., Littin, Richard H., McWha, J. A. David, and Cleary, John G. (1997). Applying Time Warp to CPU design. In *High Performance Computing Conference '97*, Bangalore, India. IEEE.

Perleberg, C. H. and Smith, A. J. (1993). Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412.

Sohi, G. S., Breach, S., and Vijaykumar, T. N. (1995). Multiscalar processors. In *22nd International Symposium on Computer Architecture (ISCA-22)*.

Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33.

Wall, Daivd W. (1991). Limits of instruction-level parallelism. In *4th International Conference on ASPLOS*, pages 176–188, New York, N.Y. ACM.