EXPANSION CACHES FOR SUPERSCALAR PROCESSORS

John D. Johnson

Technical Report No. CSL-TR-94-630

June 1994

This work was supported by NASA under NAG2-248 using facilities supplied under NASA grant NAGW 419 and equipment donations from Hewlett-Packard Company

EXPANSION CACHES FOR SUPERSCALAR PROCESSORS

by

John D. Johnson

Technical Report No. CSL-TR-94-630

June 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Superscalar implementations present increased demands on instruction caches as well as instruction decoding and issuing mechanisms leading to very complex hardware requirements. This work proposes utilizing an expanded instruction cache to reduce and simplify the complexity of hardware required to implement a superscalar machine. Trace driven simulation is used for evaluating the presented Expanded Parallel Instruction Cache (EPIC) machine and its performance is found to be comparable to a dynamically scheduled superscalar model.

Key Words and Phrases: Expansion caches, Superscalar, VLIW

Contents

1 Introduction

An increasing interest in computer organizations exploiting instruction-level parallelism (ILP) is occurring as the ability to produce larger VLSI chips increases. Research into exploiting ILP has generally developed along two major categories: machines having very long instruction words (VLIW) and machines with decode units capable of dispatching multiple scalar instructions in parallel (superscalar). This paper proposes using an expanded instruction cache to obtain the benefits of a VLIW execution engine while maintaining instruction encoding benefits found in superscalar machines.

VLIW machines are attractive because of their ability to accommodate large amounts of instruction-level parallelism with relatively simple and inexpensive control hardware [BYA93] [CNO+88]. This simple control also leads to a shorter cycle time and thus higher performance. Disadvantages of VLIW machines include increased instruction memory requirements, both in size and in bandwidth, and incompatible binary program representations for different machine configurations.

Superscalar processors [Joh91] [DA92] are better equipped to deal with variable machine configurations, have better code density, and can be binary compatible with existing scalar machines. However, a superscalar machine encounters the difficult problem of fetching and decoding multiple dependent instructions with very short latency, ideally in a single cycle. This leads to large decoding and control hardware requirements that both limits available execution parallelism and increases the cycle time.

Presented in this paper is an approach for supplying VLIW like instructions in parallel to the execution units of a machine while maintaining the advantages of sequential scalar style instruction encoding. This approach is called an Expanded Parallel Instruction Cache (EPIC) machine. It is a statically scheduled in-order execution machine whose guiding principle is simple hardware allows short cycle times. An EPIC machine organization uses an expansion cache as the mechanism to deliver instructions to the execution units. The expansion cache contains the scalar instructions along with additional fields specifying information needed for parallel execution.

Figure 1 presents a block diagram of the EPIC machine model. It shows how instructions are fetched from the main instruction memory by the Expansion Unit and stored into the Expanded Instruction Cache. Output from the expanded instruction cache is used to control multiple execution units in parallel. There are several independent and specialized execution units. These units are divided into 3 instruction types: branch instructions, load/store instructions and functional instructions.

During each execution cycle an expanded instruction is clocked into the Expanded Parallel Instruction Register. Parallel instructions contain several independent instructions and is used to control directly the parallel execution units. The independence of the individual instructions is ensured by the expansion process that initially assembled the expanded instruction. The Expanded Parallel Instruction Register is also a pipeline register as it holds the current instructions while the next instructions are being fetched from the expanded instruction cache. Bit fields in this register are connected directly to the execution units without any routing network, providing minimum delay for delivering the issued instructions to the execution units' decoders. The execution units are also pipelined, although this is not explicitly shown in figure 1.

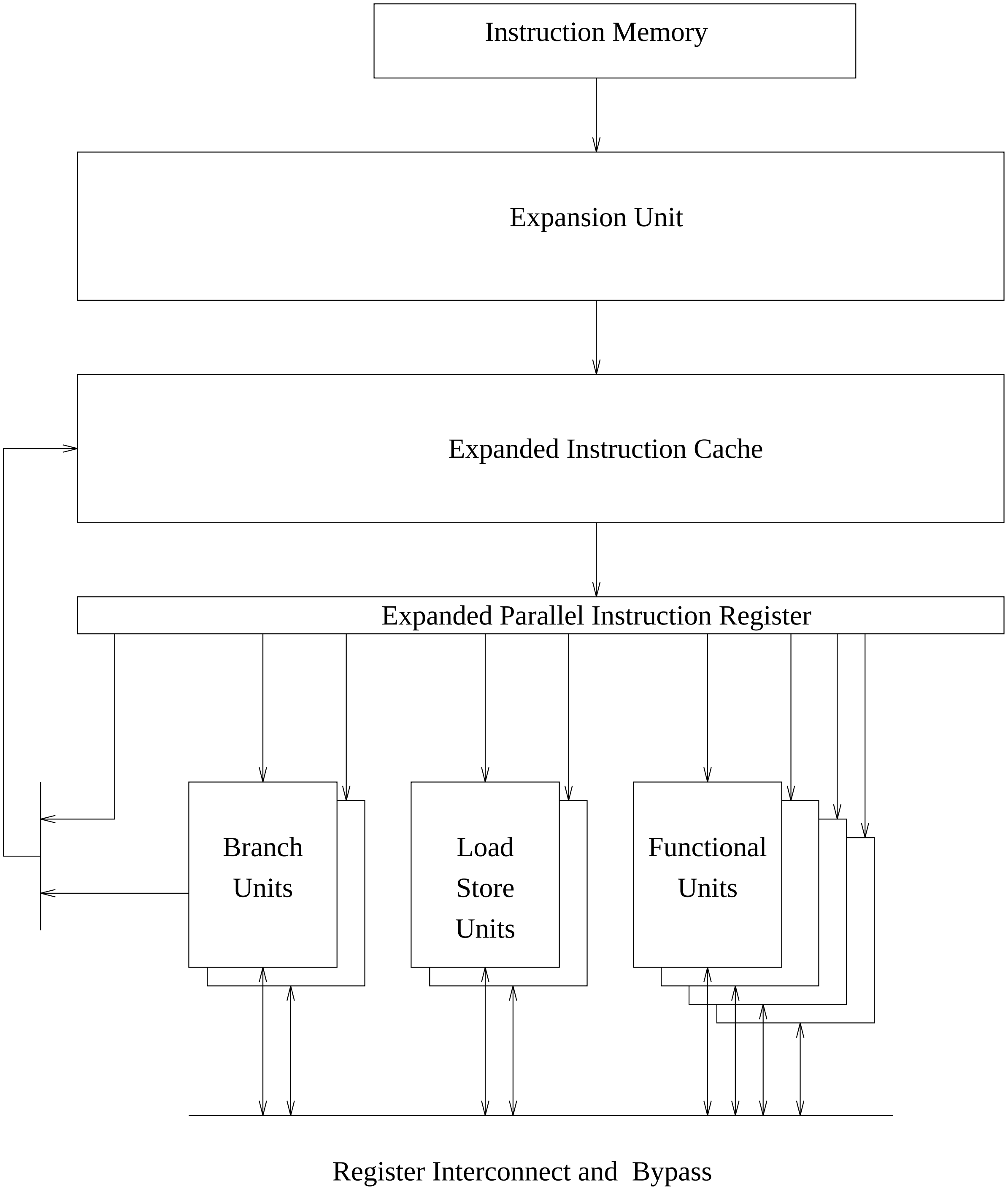
Instruction Memory
Expansion Unit
Expanded Instruction Cache
Expanded Parallel Instruction Register
Branch Units
Load Store Units
Functional Units
Register Interconnect and Bypass

Figure 1: Block Diagram of EPIC Model

Additional fields in the expanded instruction specifies the next Expanded Parallel Instruction to fetch. This is a form of branch prediction. The fetch of the next instruction group is initiated before any branches in the current instruction group are decoded.

An EPIC machine spends most of its time executing Expanded Parallel Instructions from the cache. Whenever a valid instruction is read from the Expanded Instruction Cache then the next cycle is an execution cycle. During an execution cycle data is manipulated by the execution units.

When the cache misses then the EPIC machine changes modes. After a miss, the next several cycles are expansion cycles. During expansion, instructions are read from main instruction memory, partially decoded and packed into the Expanded Parallel Instruction Cache.

The expansion unit decodes instruction at a rate of one per cycle and is therefore simpler than a superscalar decoder. A superscalar processor decoding n instructions per cycle must compare each of the n instructions with the $n-1$ prior instructions being decoded during the cycle. This results in $O(n^2)$ complexity for the dependency checking hardware. Since the EPIC model decodes only one instruction per cycle its dependency checking hardware compares a single instruction to the $n-1$ prior instructions being prepared for parallel execution. This has hardware complexity of only $O(n)$.

Note, however, the reduction in dependency checking hardware complexity achieved by the EPIC model increases decode latency. This cost is mitigated by caching the results of the expansion process in the expanded instruction cache. The increased latency is incurred only during a cache miss so it contributes only slightly to the overall execution.

The remainder of this paper is organized as follows. Section 2 discusses the issues addressed by the EPIC architecture. Section 3 presents and discusses the performance of a simulated EPIC machine and compares it to a conventional superscalar machine. Finally, section 4 offers concluding remarks and future directions.

2 Expansion Caches and Superscalar Instruction Issuing

Traditional instruction caches improve performance by reducing average memory latency. They function by providing rapid access to exact copies of some of the instructions found in main memory. This work investigates additional opportunities presented by an instruction cache when the information stored in the cache is not an exact copy of the main memory data. Ordinarily the cached representation is larger than the main memory representation, consequently the term *expansion caches* is used to describe this general caching technique.

Superscalar architectures presents increased demands on the instruction cache and instruction decoding procedures that can lead to very complex hardware. This work studies expansion caches to improve the performance of the fetch mechanism and reduce the complexity and quantity of hardware required to implement a superscalar machine.

2.1 Cache Aligning

Instruction fetch efficiency is improved if the instructions supplied by the instruction cache are aligned to the slots of the decoder. This aligning results in fewer wasted decoder slots. Assuming the decoder width is the same as the cache line width and the program counter starts at a random offset within a line then on average one half of the decode slots are wasted if alignment is not used.

The expanded instruction cache furnishes the desired alignment by employing extra bits in the cache line tags and duplicating instructions within the cache. The extra tag bits allow the expander to align any instruction to the first decoder slot. The following instructions are packed into following slots within the line, thereby achieving the desired alignment.

For example, an 8 instruction wide direct mapped cache requires adding $\log_2 8$ (=3) additional bits in the tag field of each cache line. Assume there are 2^n cache lines so n bits are needed to index the cache data array, and assume the least significant bit of the PC is numbered 0. The cache data array would be indexed by PC bits $n-1$ to 0. The instruction addressed by the PC is aligned to the first decoder slot and the following 7 instructions are aligned to the following 7 decoder slots. A conventional direct mapped cache uses PC bits $n+2$ to 3 to index the data array, while bits 2 to 0 are used control the shifter that delivers the requisite instructions to the decode slots.

2.2 Branch Prediction

Accurate high speed branch prediction is an important component of high performance machines. Typical workstation code contains about one branch in every 5 or 6 instructions. With branches being this frequent it is imperative that superscalar machines predict the outcomes of branches during instruction fetch, without waiting for the execution units to generate the correct target address.

In the EPIC model each cache line is extended to include an additional field called the *successor address* field. This successor address field provides information for instruction sequencing and branch prediction. This field directly specifies the next cache line to fetch. Fetching the next group of instructions is initiated as soon as the current group is latched into the expanded instruction register.

During the expansion process, the expander predicts a branch's target. Static branch prediction based on compiler supplied information is used for the results presented in this report but other static or dynamic methods are possible. The predicted target address and speculative execution information, as explained in the next section, are used to compute the successor address field.

A high performance implementation requires the successor address field to be the same as the width of the PC, normally 30 bits. It is possible to reduce the width of the successor index field to be the number of bits needed to index the cache, recovering the remaining address bits from the cache tag at the successor index. However, this delays detection of a miss predicted branch by at least one cycle.

One additional bit is required for each branch packed into the expanded instruction. This bit specifies the predicted taken/not-taken decision made at expansion time, thereby permitting a branch execution unit to determine if the branch was correctly predicted.

A branch unit executes a conditional branch by comparing registers or condition codes as specified by the opcode and determining if the branch direction matches the saved prediction bit. If the prediction is correct the execution of the branch instruction is complete. The target instruction is either merged in with the expanded instruction, as explained by the next section, or it is at the successor index of this expanded instruction and is being fetched this cycle.

2.3 Instruction Run Merging

Instruction run merging is the process of simultaneously providing the decoder with instructions from both before and after a branch instruction. Merging increases decoder efficiency because it fills decoder slots after a branch with probably useful instructions. Merging is accomplished in EPIC by the expander continuing instruction expansion after it encounters and predicts a branch. The expander fetches instructions from the predicted target of the branch and packs these instructions into the expanded instruction register until this register is filled.

During execution time, the instructions packed after a conditional branch are speculatively executed in parallel with the branch. If the branch is a conditional branch then these instructions must be flagged as being speculatively executed, but no other special handling is needed during execution of these instructions. The special handling for predicting the branch direction and the non-sequential fetch from main memory occurs only during expansion time, when the demand for high speed alignment and decoding is less stringent.

If a branch is incorrectly predicted then the branch execution unit must arrange to cancel any result writes by speculative instructions initiated during the same cycle. It must also prevent the expanded instruction currently being fetched this cycle from being executed next cycle. The branch unit sends the correct target execution address to the expanded instruction cache and this expanded instruction is fetched during the next cycle.

No special reorder buffers or other hardware is needed to recover from a miss predicted branch because all speculative instructions are issued during the same cycle as the branch they depend upon. Thus speculative instructions can be canceled before they reach the result write stage. This eliminates the need for special hardware result buffering of speculative instructions and simplifies the machine.

2.4 Alignment to Execution Units

In many superscalar implementations, the routing network delivering instructions from the decoders to the proper execution units requires significant time and may even be a component of the critical path cycle time. Each individual decoder must be able to route the instruction it is decoding to any of execution units. If there are n decoders and m execution units then the hardware for the routing network has complexity $O(n \times m)$.

For the EPIC machine a different approach to instruction routing is used. Instead of routing the instructions after they are fetched from the cache, the instructions are aligned to the appropriate unit when they are inserted into the cache.

This simple direct control arrangement is motivated by the desire to allow a very short cycle time when this machine is implemented in hardware. No complicated and communication intensive resource allocation logic is needed during the interpretation of the wide expanded parallel instruction. The bits from the expanded instruction directly drive the execution units, thereby requiring no additional gate delays to select a highest priority instruction and shorting the cycle time.

Aligning instructions to the appropriate unit is the responsibility of the expansion unit. It decodes each instruction it fetches and routes it to the appropriate slot in the expanded instruction being built. This causes the instructions in the expanded instruction to be arranged in a different order than their original program order. Rearrangement is legal because the expander ensures all instructions entered into the line are independent.

Additional bits are needed within the expanded instruction to reconstruct original program order information. The original order is required to identify which instructions follow a miss predicted branch so all instructions following such a branch can have their result writes disabled. The original program order is also needed to generate the PC appropriate for an exception such as a page fault.

The number of additional bits needed to specify the original program order of individual instructions within a line depends on order reconstruction logic. If n individual instructions can be packed into an expanded instruction line there are $n!$ ways this packing can be performed. The minimum bits required to specify this number of packing is $\lceil \log_2(n!) \rceil$, which is 16 bits for an 8 instruction line. If this logic requires more time to reconstruct the instruction order information than conditional branch requires to execute, then it is appropriate to use a less encoded ordering information field. The simplest form is just a bit field attached to each individual instruction specifying its original ordering. This simple encoding uses eight 3 bit fields, for a total of 24 bits in the expanded instruction for original program order information.

When the expander is unable to pack the instruction being expanded into the expanded instruction line because of limited execution units it could simply fill the remaining slots with NOPs. Each expanded instruction would have individual instructions aligned to the required execution units, and achieve the goal of very simple control hardware. However, this is inefficient use of cache memory. A more efficient method of packing the expanded instruction that still preserves the direct execution unit alignment is presented in the next section.

2.5 Expanded Cache Efficiency

Cache efficiency is an important issue for expanded instruction cache machines. The efficiency of the instruction cache is reduced both by the duplication of instructions within the cache and the inserting of NOPs for overcoming data hazards and providing alignment.

Experiments show 10% to 20% increase in required cache size because of duplication of instructions. However, the partial filling of cache lines due to data dependencies leads to a much larger efficiency penalty. The expanded stops packing instruction into a group when it finds a data dependency or resource limitation. On average about 30% of the slots are filled because of data dependency and execution unit alignment requirements for an 8 instruction line. If this factor of 3 inefficiency in cache size could not be mitigated, an expanded instruction cache machine probably would not be competitive with other machine organizations.

Cycle tagging is the method used in EPIC to increase cache efficiency. Instead of having each expanded instruction specify only one cycle's worth of execution, each expanded instruction specifies several cycles worth of execution. Each instruction is tagged with the cycle it is to execute.

Alignment to execution units is maintained in the cycle tagged structure. The expander aligns the first cycle's instructions to the appropriate execution units. This achieves the required alignment for the first cycle after a fetch from the expanded cache. Instructions tagged for the second and following execution cycles are placed by the expander into the first available slot in the expanded instruction. This achieves high utilization of the cache line.

Having the first cycle's instructions already aligned within the expanded instruction meets the goal of direct execution unit control for the first cycle. During the first cycle there is time to route the second cycle's instructions into position for direct control of the appropriate execution units during the second cycle. Instructions for the third cycle are positioned during the second and so on. Aligned first cycle instructions with routed second cycle and beyond instruction accomplishes the simple direct control of execution unit required to minimize cycle time.

Cycle tagging achieves 80% to 90% utilization of the cache lines. Performance considerations limit utilization. If the lines were filled to 100% of capacity then many instruction packets would be split between cache lines and parallelism would be lost. When the cache line is filled to capacity then instruction duplication also increases, negating the efficiency advantage one is attempting to gain.

2.6 Interlocks and Code Scheduling

The philosophy of the EPIC machine is to have the simplest possible hardware in the instruction issue unit. Ideally, there would be no interlock hardware at all, as is the case for some VLIW machines. However, since the code compiled for EPIC is not cognizant of the exact packing and cache misses occurring during execution, it is not possible to schedule code that is free of multi-cycle data hazards, so some interlock mechanism is required.

A simple register scoreboard interlock mechanism is used in the EPIC machine. There is one "busy" bit for each machine register in this scoreboard. Decoding a multiple cycle instruction sets the scoreboard busy bit corresponding to the destination register. This busy bit is cleared at the end of the cycle just before the instruction's result is ready. For the one cycle load delay instructions used in the EPIC this bit is cleared the next cycle.

The execution unit decoder for each individual instruction within an expanded instruction inspects scoreboard busy bits corresponding to all source operands. If any bit is set then not all the instruction's source operands are ready. A signal is generated stalling all components of the expanded instruction line being decoded. Next cycle the scoreboard bits are read again and if they are all clear then all instructions in the expanded instruction register will be issued in parallel.

Stalling of all instructions within the expanded instruction ensures that in-order issue is maintained. This in-order issue policy greatly simplifies the hardware required to implement the parallel execution units.

Stalling of all individual instruction within an expanded instruction impacts the code scheduling for EPIC. There are cases when it is desirable *not* to pack an independent instruction that will be stalled because of data dependency into the current expanded instruction line. It is better to place it into the next line, thereby allowing the other independent instructions to proceed. To accommodate this form of scheduling an *align* instruction is used. The align instruction informs the expansion process to stop filling the current expanded instruction. The expanded instruction will then be executed. When needed, the expansion process will restart after the align instruction and fill the next line. The align instruction is information used only by the expander, it is never entered into the expanded instruction cache.

3 Experiments

A number of experimental simulations were conducted to ascertain the performance of the EPIC model. Since the major emphasis of this study is on the instruction fetch and decode aspects of superscalar machines, the simulated machines contain generous execution resources. This causes the performance to be limited by the fetch, decode and issue components of the machine and rather than the execution components.

The EPIC simulation results are most relevant when they are compared to other machine organizations using with similar configurations. This is accomplished by conducting two different simulations driven by the same compiled code and instruction traces. The first simulator is for the proposed EPIC reduced complexity superscalar machine. The second simulator is a performance reference model and has the instruction fetch, decode, issue and reorder buffer hardware configuration published by Mike Johnson [Joh91].

3.1 Methodology and Benchmarks

Trace driven methodology produces the results of this study. The benchmarks are instrumented and executed to produce trace files. These trace files are used to drive the performance simulators.

Table 1 describes the six benchmarks included in this study. These benchmarks mimic an execution profile similar to a typical workstation environment. Five of the benchmark programs are also used in the SPEC benchmark suite, although the version of these programs and input data sets used here are not the same as those used in the SPEC suite. All the benchmarks are written in the C language.

Benchmark	Description
`compress`	Lempel-Ziv compression on a 150KB tar file.
`espresso`	Boolean expression minimizer reducing a 14-bit input, 8-bit output PLA.
`fft`	Fast Fourier transform – 1024 1-D fft
`gcc1`	Gnu C version 1.36 compiling (and optimizing) to assembly 1500 lines of C.
`spice3`	Circuit simulation of a Schottky TTL edge-triggered register.
`tex`	Document preparation system formatting a 14 page technical report.

Table 1: Description of the Benchmarks

3.2 Machine Configurations

Table 3 presents the two machine configurations being evaluated These configurations provide enough resources so that the machines can exploit almost all the instruction-level parallelism available in the benchmarks.

The EPIC machine has the form of figure 1, except that the expanded instruction cache is divided into a first level and a second level expanded instruction cache.

A two level expanded instruction cache plays an important role when the first level cache is small. Since an expanded instruction cache miss has a larger miss penalty than a traditional cache, it is important that the miss rate is low for the overall cache structure. This is accomplished by the large second level cache. Also, since an expanded instruction cache is not as efficient in terms of storage as a normal cache, it is important the miss service time for the first level cache be as small as possible.

The expansion unit fills the 2nd level instruction cache. The 1st level expanded instruction cache is filled from the 2nd level instruction cache with a 4 cycle penalty for each miss. Second level misses incur a 16 cycle main memory latency. Expansion time is also added to 2nd level miss service time. The 2nd level cache is modeled as an infinite size cache, a reasonable model for an off chip cache. The rest of the EPIC machine is as described in section 2.

The Reorder Buffer machine is a superscalar machine with an organization modeled after the work

Benchmark	Static Size (bytes)	Dynamic Instructions
compress	6,760	13,252,875
espresso	96,732	153,611,785
fft	1,456	6,692,459
gcc1	540,276	42,774,378
spice3	469,356	151,890,201
tex	161,012	72,706,435

Table 2: Benchmark Sizes and Instruction Counts

	EPIC	Reorder Buf.
Issue Policy	in-order	out-of-order
Interlock Mechanism	scoreboard	result tags
Register Renaming	none	reorder buf.
Cache Line Size (inst.)	8	8
Expansion Pipe Stages	3	n/a
Reservation Stations	n/a	4, 8 for L/S
Reorder Buffer Entries	n/a	16
Decode Stages	1	2
Miss Pred. Brn Penalty	1	3
Branch Units	2	2
Functional Units	4	4
Load/Store Units	2	2
Load Delay (cycles)	1	1
1st Level Cache Miss	4	4
2nd Level I-Cache	infinite	infinite
Data Cache	infinite	infinite

Table 3: Simulated Machine Configurations

of Mike Johnson's "Standard Processor" [Joh91]. The philosophy of this Reorder Buffer model is that finding as much instruction-level parallelism as possible is more important than reducing the amount of communication and resource allocation logic. Whether this additional logic increases the cycle time is dependent upon many implementation factors. For the performance comparison graphs in this study it is assumed that the EPIC model and the Reorder Buffer model have the same cycle time.

The Reorder Buffer machine attempts to yield the best superscalar performance by using out-of-order issue. Reservation stations between the instruction decoder and execution units buffer some instructions while others are issued. The register file contains the in-order state while the reorder buffer contains the lookahead state.

Instruction decoding in the Reorder Buffer machine must access both the register file and the reorder buffer as well as route the instructions to the appropriate reservation stations. The Reorder Buffer machine is modeled with a two cycle decode process to accommodate these complications. The EPIC machine has a single cycle decode because it uses in-order issue and execution unit alignment to eliminate the reorder buffer and the instruction routing.

Recovering from a miss predicted branch requires only one cycle in the EPIC machine because it has a single decode stage. On the Reorder Buffer machine, two cycles of penalty are charged for a miss predicted branch because of its two cycle decode. The Reorder Buffer machine must also flush all speculative instructions out of its reservation stations, load pipelines, result buses and the reorder buffer after a miss predicted branch. If it is required to perform this flushing during the same cycle as issuing the correct target instructions are issued then additional addressing ports are required. Consequently an additional flush cycle is required for this machine. The extra decode cycle and extra flush cycle cause the Reorder Buffer machine to have a total miss predicted branch penalty of 3 cycles.

Overall, the configurations of the two machines are as similar as possible. Both machines are assumed to have the same cycle time and have the same execution units. Main memory bandwidth, latency and 1st level instruction cache miss penalties are all the same. These similarities in configuration cause the performance differences reported in the next section to be almost completely due to the differences in fetch and decode strategy.

3.3 Results

Simulators model the execution of the EPIC and Reorder Buffer machines. Trace driven simulation is used to evaluate their performance. Benchmarks are compiled and optimized using the DEC Ultrix C compiler and the Ucode output captured. This Ucode was then scheduled for the parallel execution machine using a basic block list scheduling algorithm. The same scheduled code is used as the input to both the EPIC and Reorder Buffer machines.

Figure 3.3 presents the performance of the two machines for each benchmark. The vertical axis of the graphs is the average instructions per cycle for each benchmark. The horizontal axis is the 1st level instruction cache size in bytes. The size plotted is the amount of storage available for instruction data. Overhead for the tags, successor index, ordering and align bits is not included. The actual width of an EPIC cache line is about 16% wider than a Reorder Buffer cache line.

The graphs show the EPIC machine almost always achieves better performance than the much more complicated Reorder Buffer machine. For most of the benchmarks both machines are able to execute about 1.5 instructions per cycle. The FFT benchmark is somewhat better at 2.9 and 2.6 instructions per cycle for the EPIC and Reorder Buffer machines. This is due to FFT's large basic blocks and the ability of the compiler to effectively schedule code for the 8 instructions per cycle peak bandwidth of the machines.

With first level cache sizes of 64K and above, the EPIC machine is performing moderately better than the Reorder Buffer machine. Small first level cache sizes and large benchmarks reduce performance of the EPIC machine because efficient usage of cache storage is more important in this cache size range.

It is surprising the EPIC machine shows any performance advantage at all. EPIC's organization is intended to improve performance by permitting a shorter cycle time. For the comparison here the

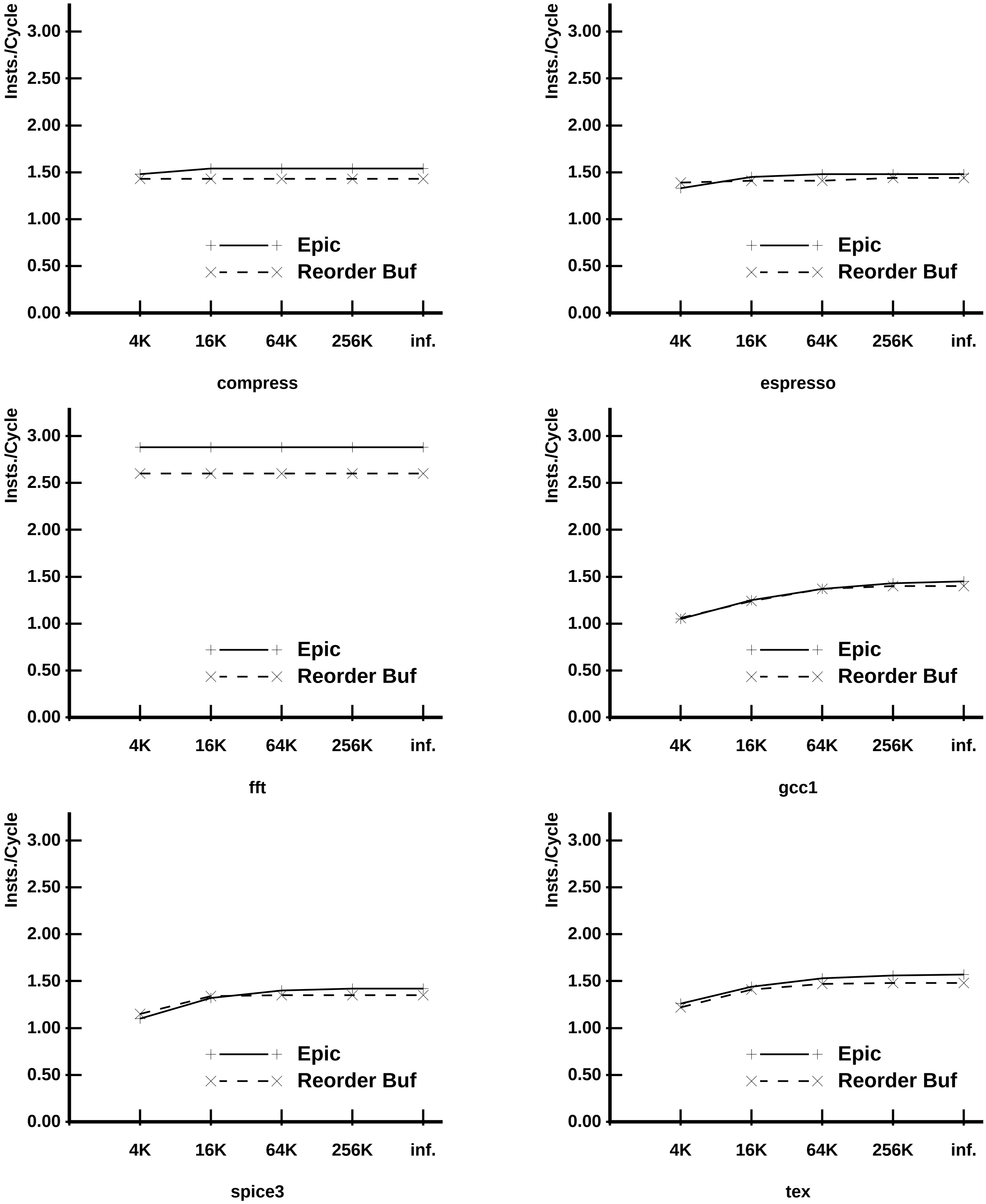

Figure 2: Performance in Instruction per Cycles *vs.* Level 1 Cache Size

cycle time for both machines is the same and EPIC still achieves comparable performance.

The Reorder Buffer machine is organized to take advantage of almost all the instruction-level parallelism within the limits of its instruction window. As soon as an instruction has its operands available it is issued out of a reservation station, so no execution unit is ever idle while there are instructions ready to execute.

The EPIC machine on the other hand stalls up to seven other independent instructions if an instruction can not be issued because an operand is not available. EPIC's in-order issue intentionally does not utilize all available ILP in order to simplify its hardware. In-order issue eliminates the need for the reorder buffer and reservation stations. These simplifications allow shorter cycle times for the EPIC machine.

4 Conclusions

We have demonstrated an expanded instruction cache structure can be used to reduce the complexity of the hardware required to implement a superscalar machine without reducing its performance in terms of instructions per cycle. The expanded instruction cache is able to improve decoder efficiency in the three areas of cache alignment, branch prediction and instruction run merging. The expanded instruction cache is also able to align instructions with the required execution units without requiring a time consuming routing network. Taken together these improvements compensate the slight loss in detecting available instruction-level parallelism and permit the EPIC machine to have improved performance relative to a Reorder Buffer machine, even when both machines have the same cycle time.

Cache storage efficiency issues raised by expanded instruction caching are addressed by the cycle tagging and two level cache structures. Cycle tagging achieves an overall utilization of cache storage in the 90% range and the two level cache structure achieves a low average miss service time.

Expanded instruction caching is a promising organization for future superscalar machines. We are continuing our studies into the features presented here along with other refinements and performance improvements.

References

[BYA93] G.R. Beck, D.W.L. Yen, and T.L. Anderson. The Cydra 5 mini-supercomputer: architectural and implementation. *The Journal of Supercomputing*, 7(1):143–180, May 1993.

[CNO+88] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[DA92] K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 2(2):40–63, April 1992.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.