View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Reducing Fetch Architecture Complexity Using Procedure Inlining

Oliverio J. Santana, Alex Ramirez, and Mateo Valero, Fellow, IEEE
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
email:{*osantana,aramirez,mateo*}@ac.upc.es

Abstract

Fetch engine performance is seriously limited by the branch prediction table access latency. This fact has lead to the development of hardware mechanisms, like prediction overriding, aimed to tolerate this latency. However, prediction overriding requires additional support and recovery mechanisms, which increases the fetch architecture complexity.

In this paper, we show that this increase in complexity can be avoided if the interaction between the fetch architecture and software code optimizations is taken into account. We use aggressive procedure inlining to generate long streams of instructions that are used by the fetch engine as the basic prediction unit. We call instruction stream to a sequence of instructions from the target of a taken branch to the next taken branch.

These instruction streams are long enough to feed the execution engine with instructions during multiple cycles, while a new stream prediction is being generated, and thus hiding the prediction table access latency. Our results show that the length of instruction streams compensates the increase in the instruction cache miss rate caused by inlining. We show that, using procedure inlining, the need for a prediction overriding mechanism is avoided, reducing the fetch engine complexity.

1 Introduction

High performance superscalar processors require high fetch bandwidth to exploit all the available instruction-level parallelism. The development of accurate branch prediction mechanisms has provided important improvements in the fetch engine performance. However, it has also increased the fetch architecture complexity. Our approach to achieve high fetch bandwidth, while maintaining the complexity under control, is the stream fetch engine [14].

This fetch engine design is based on the next stream predictor, an accurate branch prediction mechanism which uses instruction streams as the basic prediction unit. We call stream to a sequence of instructions from the target of a taken branch to the next taken branch, potentially containing multiple basic blocks. Although a fetch engine based on streams is not able to fetch instructions beyond a taken branch in a single cycle, streams are long enough to provide a high fetch bandwidth. In addition, since streams are sequentially stored in the instruction cache, the stream fetch engine does not need a special-purpose storage, nor a complex dynamic building engine.

However, taking into account current technology trends, accurate branch prediction is not enough. The continuous increase in processor clock frequency, as well as the larger wire delays caused by modern technologies, prevent branch prediction tables from being accessed in a single cycle [1, 8]. This limits fetch engine performance because each branch prediction depends on the previous one, that is, the target address of a branch prediction is the starting address of the following one.

A common solution for this problem is the prediction overriding technique [8, 19]. A small and fast predictor is used to obtain a first prediction in a single cycle. A slower but more accurate predictor provides a new prediction some cycles later, overriding the first prediction if they differ. This mechanism partially hides the branch predictor access latency. However, it also causes an increase in the fetch architecture complexity, since prediction overriding requires a complex recovery mechanism to discard the wrong speculative work based on overridden predictions.

An alternative to the overriding mechanism is using long basic prediction units. A stream prediction contains enough instructions to feed the execution engine during multiple cycles [14]. Therefore, the longer a stream is, the more cycles the execution engine will be busy without requiring a new prediction. If streams are long enough, the execution engine of the processor can be kept busy during multiple cycles while a new prediction is being generated. Overlapping the

Proceedings of the Eighth Workshop on Interaction between Compilers and Computer Architectures (INTERACT'04)
0-7695-2061-8/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

execution of a prediction with the generation of the following prediction allows to partially hide the access delay of this second prediction, removing the need of an overriding mechanism, and thus reducing the fetch engine complexity.

Since instruction streams are limited by taken branches, the best way to obtain longer streams is removing taken branches through code optimizations. Code layout optimizations have a beneficial effect on the length of instruction streams [14]. These optimizations try to map together those basic blocks which are frequently executed as a sequence. Therefore, most conditional branches in optimized code are not taken, enlarging instruction streams. However, code layout optimizations are not enough for the stream fetch engine to completely overcome the need of an overriding mechanism [18].

In this paper, we show that more aggressive optimizations provide longer instruction streams, increasing the stream predictor ability of tolerating the access latency. In particular, we focus on procedure inlining. This optimization replaces a procedure call by the procedure itself, removing the call and return instructions. Since both procedure calls and return instructions are taken branches, procedure inlining involves an increase in the length of instruction streams. In addition, removing procedure boundaries enables more opportunities for code optimizations.

We use the ALTO [10] tool to perform an aggressive procedure inlining. The main drawback of this optimization is that it increases the code size, and thus increases the number of instruction cache misses. However, the beneficial effect of longer streams compensates this increase in the instruction cache miss rate. Moreover, this aggressive procedure inlining optimization provides streams long enough to hide the branch predictor access latency without requiring an overriding mechanism, reducing the fetch architecture complexity. These results illustrate how taking advantage of code optimizations allows the design of high performance fetch architectures with a low cost and complexity.

The remainder of this paper is organized as follows. Section 2 presents previous related work. Section 3 explains our experimental methodology. Section 4 describes the effect that procedure inlining has on the length of instruction streams. Section 5 shows that prediction overriding can be avoided thanks to procedure inlining. Finally, Section 6 presents our concluding remarks.

2 Related Work

Procedure inlining is a frequently used code optimization. Allen and Johnson [2] describe a procedure inliner for C programs. However, they consider that only small procedures should be inlined to avoid an increase in the number of instruction cache misses. Hwu and Chang [7] present profile-driven algorithms for applying inlining and code reordering. The profile information is used to decide whether inlining a procedure will be beneficial for the program execution, allowing to inline bigger procedures. In addition, reordering the program code is an effective technique to alleviate the increase in the instruction cache misses caused by inlining big procedures.

Ayers et al. [4] describe an aggressive inliner, based on profile information, that is able to inline procedures at almost any call site without restriction. Their results show that aggressive inlining can provide important performance improvements in some benchmarks. Likewise, the ALTO [10] optimizer is able to aggressively inline procedures, using profile-based code reordering to reduce the negative effects of inlining on the instruction cache. Aydin and Kaeli [3] take this one step further implementing cache line coloring algorithms. They use ALTO to aggressively inline procedures, showing that cache line coloring is beneficial for reducing the negative impact of inlining on the instruction cache miss rate.

As Aydin and Kaeli, we use ALTO to perform aggressive procedure inlining. However, we focus our research on the length of basic prediction units instead of the instruction cache miss rate. Long prediction units, for example instruction streams [14], allow to tolerate the prediction table access latency by feeding the execution engine with instructions during multiple cycles. Inlining enlarges instruction streams by removing dynamic taken branches, that is, function calls and returns. Our claim is that the increase in the number of instruction cache misses caused by inlining is compensated by the increase in the length of instruction streams and the additional optimizations enabled by inlining, improving the overall processor performance.

In particular, adequately balancing the number of instruction cache misses and the length of instruction streams, we show that the next stream predictor can tolerate the prediction table access latency without requiring additional hardware mechanisms, like prediction overriding [8, 19]. This mechanism provides two predictions, a first prediction coming from a fast branch predictor, and a second prediction coming from a slower, but more accurate predictor. When a prediction should be generated, the first prediction is used while the second one is still being calculated. Once the second prediction is obtained, it overrides the first one if they differ, since the second predictor is considered to be the most accurate.

The advantage of prediction overriding is that it benefits those binaries that have already been compiled without code optimizations specifically devoted to hide the prediction table access latency. However, the main problem of prediction overriding is that it requires an important increase in the fetch engine complexity. An overriding mechanism requires a fast branch predictor to obtain a prediction each cycle. This prediction should be stored for being compared with the main prediction. Some cycles later, when the main

prediction is generated, the fetch engine should determine whether the first prediction is correct or not. If the first prediction is wrong, all the speculative work done based on it should be discarded. Therefore, the processor should track which instructions depend on each prediction done in order to allow the recovery process.

In this way, the stream fetch engine reduces the fetch engine complexity by taking advantage of code optimizations. There are other fetch architectures able to exploit code optimizations. The trace cache architecture [13, 16, 5] provides high fetch performance by buffering and reusing dynamic instruction traces. These traces are long enough to partially hide the prediction table access latency [18]. The rePLay microarchitecture [12] uses a front-end derived from the trace cache to generate even longer traces. These traces, called frames, are built making an extensive use of the branch promotion technique [11] and later applying dynamic optimizations.

The main advantage of these approaches over the stream fetch engine is that they should not stop fetching instructions when a taken branch is found. In addition, the stream fetch engine is not able to make dynamic optimizations, like rePLay does. A performance comparison against these mechanism is out of the scope of this paper. However, it is important to note that, since instruction streams are consecutively stored in the instruction cache, the stream fetch engine does not require an additional special-purpose storage, nor a dynamic building mechanism like the trace cache and rePLay. This involves an important reduction in the fetch engine complexity, which is the main objective of this paper.

3 Experimental Methodology

The results in this paper have been obtained using trace driven simulation of a superscalar processor. Our simulator uses a static basic block dictionary to allow simulating the effect of wrong path execution. This model includes the simulation of wrong speculative predictor history updates, as well as the possible interference and prefetching effects on the instruction cache.

We simulate the SPECint2000 benchmarks optimized using ALTO [10]. First, the benchmarks were compiled with the Compaq C V5.8-015 compiler on Compaq UNIX V4.0, using the *-O2* optimization level without inlining. These binaries were later optimized with ALTO[1] using profile information collected by the pixie V5.2 tool using the *train* input set. We use the ALTO tool to generate our baseline binaries, optimized without inlining, as well as several sets of binaries with different levels of inlining.

Since we evaluate different sets of binaries, we simulate all benchmarks until completion to assure a fair comparison. In order to explore a wide range of setups and binaries, we have chosen benchmark inputs from the *MinneSPEC* [9] input set. This input set has been specially designed to facilitate efficient simulations, avoiding excessively large simulation times. However, not all SPECint2000 benchmarks had a *MinneSPEC* input available when we selected our input set. In addition, we have discarded *MinneSPEC* inputs derived from the *train* input set. For those benchmarks that do not have an adequate *MinneSPEC* input available, we selected an input from the official *test* input set. Our benchmark input set is shown in Table 1.

[1] We do not simulate the benchmark *252.eon* because our ALTO version is unable to optimize it.

Table 1. Benchmark suite used and the corresponding input set.

benchmark	input	input set
164.gzip	input.random	minnespec
175.vpr	place	minnespec
176.gcc	cccp.s	test
181.mcf	inp.in	minnespec
186.crafty	crafty.in	test
197.parser	red.in	minnespec
253.perlbmk	makerand.pl	minnespec
254.gap	test.in	test
255.vortex	persons.1k	test
256.bzip2	input.source	minnespec
300.twolf	test	test

3.1 Processor Setup

In order to analyze the effect of different processor widths, we simulate two processor setups: a 4-wide and an 8-wide superscalar processor. The main values of these setups are shown in Table 2. Our simulator models the stream fetch engine [14], shown in Figure 1. This fetch model is based on a specialized branch predictor, the next stream predictor, which provides stream-level granularity, that is, it steps trough the code one stream at a time.

The next stream predictor access is decoupled from the instruction cache access using a fetch target queue (FTQ) [15]. The stream predictor generates requests which are stored in the FTQ. These requests are used to drive the instruction cache, obtain a line from it, and select which instructions from the line should be executed. Our instruction cache setup uses wide cache lines, that is, 4-times the processor fetch width, as described in [14]. We vary the total instruction cache hardware budget from 8KB to 64KB in order to explore the impact of inlining on different cache sizes.

3.2 Stream Predictor Setup

In this paper, we evaluate the next stream predictor using realistic prediction table access latencies. We have measured the access time for the stream predictor tables using the CACTI 3.0 tool [20], a detailed wire and transistor structure model of cache memories. Data we have obtained cor-

0-7695-2061-8/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

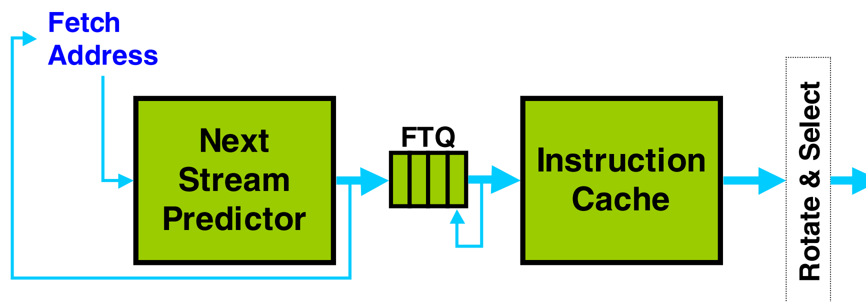


Figure 1. Stream fetch engine.

Table 2. Configuration of the simulated processors.

	4-wide processor	8-wide processor
fetch width	4 instructions	8 instructions
rename/commit width	4 instructions	8 instructions
integer issue width	4 instructions	8 instructions
floating point issue width	4 instructions	8 instructions
load/store issue width	2 instructions	4 instructions
fetch target queue	4 entries	4 entries
instruction fetch queue	16 entries	32 entries
integer issue queue	32 entries	64 entries
floating point issue queue	32 entries	64 entries
load/store issue queue	32 entries	64 entries
reorder buffer	128 entries	256 entries
integer registers	96	160
floating point registers	96	160
level-1 instruction cache	8/16/32/64 KB, 2-way associative, (4*fetch width) byte block	
level-1 data cache	64 KB, 2-way associative, 64 byte block	
level-2 unified cache	1 MB, 4-way associative, 128 byte block, 10 cycle latency	
main memory latency	100 cycles	

responds to a 0.10μm technology. For translating the access time from nanoseconds to cycles, we assumed an aggressive 8 fan-out-of-four delays clock period, that is, a 3.47 GHz clock frequency as reported in [1]. It has been claimed by Hrishikesh et al. [6] that 8 fan-out-of-four delays is the optimal clock period for integer benchmarks in a high performance processor implemented in 0.10μm technology.

In order to find the optimal stream predictor setup, we have evaluated the stream fetch engine varying the predictor size from small and fast tables to big and slow tables. Figure 2 shows the stream prediction table access time obtained using CACTI. We have measured the access time for prediction tables ranging from 32 to 4096 entry tables. These tables are assumed to be 4-way associative because 2-way associative tables require the same number of cycles to be accessed in the evaluated setups, while direct mapped tables provide a poor performance.

Taking into account realistic table access latencies, the best performance is achieved using the larger three cycle latency tables [17]. Although bigger predictors are slightly more accurate, their increased access delay harms processor performance. On the other hand, predictors with a lower latency are too small and provide a poor performance. Therefore, we have chosen to evaluate the stream predictor using the bigger table that can be accessed in three cycles, that is, 4096 entries.

Table 3. Configuration of the simulated stream predictor.

next stream predictor	1-cycle predictor
1024 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 10-2-4-10	32 entry, 1-way, predictor DOLC 0-0-0-5

Table 3 shows the configuration of the stream predictor simulated. The stream predictor actually has two prediction tables [14], a first level indexed using the stream starting address and a second level indexed using correlation with the starting address of previous streams. The first level is smaller than the second one because a larger first level table does not provide a significant improvement in prediction accuracy. We have also explored a wide range of DOLC history register [14] configurations, and selected the best one.

Since we use prediction overriding as comparison baseline, we also model a prediction overriding mechanism [8, 19]. A small stream predictor, which is supposed to be

0-7695-2061-8/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

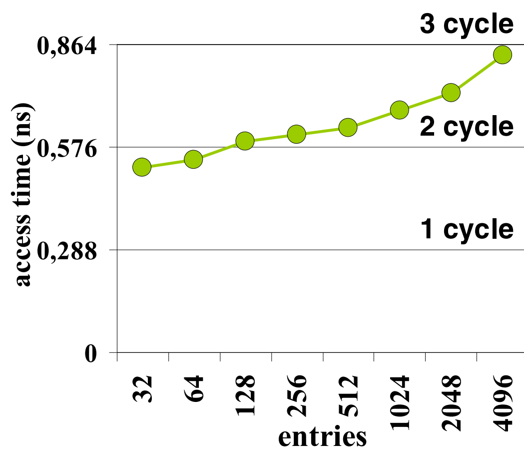


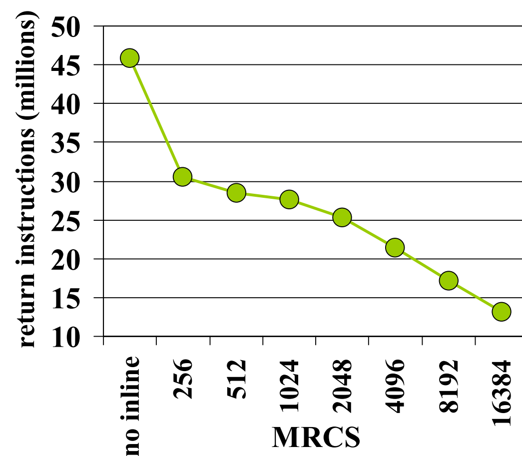
Figure 2. Access time to stream prediction tables in 0.10μm technology with a 3.47 GHz clock frequency.

implemented using very fast hardware, provides a stream prediction in a single cycle. Three cycles later, when the main stream prediction is generated, both predictions are compared. If they differ, the main prediction overrides the first one, discarding all the speculative work based on it. Table 3 shows the setup of the single-cycle predictor used by the overriding mechanism.

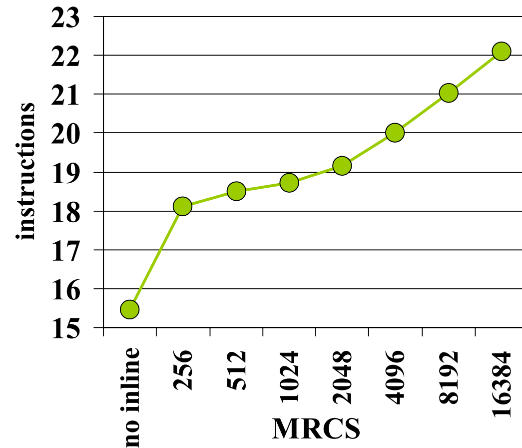
4 The Impact of Inlining on the Length of Instruction Streams

The ALTO [10] optimizer is able to perform an aggressive procedure inlining. This inlining optimization is mainly controlled by the maximum resultant code size (MRCS), that is, the maximum number of instructions that an inlined portion of code should have. A procedure is never inlined if the resultant code size is higher than MRCS. If the inlined procedure is called from a loop, then the number of instructions belonging to the loop plus the number of instructions belonging to the inlined procedure cannot be higher than MRCS. Otherwise, the number of instructions belonging to the caller procedure plus the number of instructions belonging to the inlined procedure cannot be higher than MRCS.

The higher the MRCS value is, the more aggressive is the procedure inlining performed. As a measure of procedure inlining effectivity, Figure 3.a shows the average number of return instructions. This number is equivalent to the average number of executed procedures. We evaluate the MRCS parameter varying from 256 to 16384 instructions, and compare it against our baseline code, that is, code optimized using ALTO without inlining. Higher values of MRCS involve a more aggressive inlining, and thus a higher reduction in the total number of return instructions. The more



(a) return instructions



(b) stream length

Figure 3. Average number of return instructions and average stream length varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

aggressive inlining provides a reduction over 70% of return instructions against the code optimized without inlining.

The aggressivity of procedure inlining has a direct impact on the length of instruction streams, as shown in Figure 3.b. By definition, instruction streams are limited by taken branches. Procedure inlining removes a big amount of taken branches: function calls and return instructions. The reduction in the number of taken branches involves an enlargement of instruction streams. This enlargement is limited by the removal of instructions associated with the procedure call overhead. Nevertheless, the overall effect is that the more aggressive the inlining is, the longer instruction streams are.

However, this enlargement of instruction streams is not for free. Aggressive inlining duplicates big amounts of the program code, increasing the number of instruction cache misses. Figure 4 shows the average number of instruction cache misses varying the MRCS value from 256 to 16384

0-7695-2061-8/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

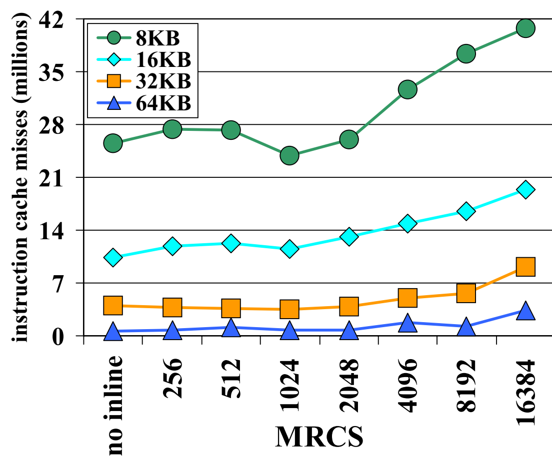


Figure 4. Average instruction cache misses varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

instructions, and the total instruction cache hardware budget from 8KB to 64KB. Although, in general, more aggressive inlining involves a higher number of instruction cache misses, this is not always a direct relationship. For example, the 8KB instruction cache has a lower number of misses using a 1024-instruction MRCS value than using a 512-instruction MRCS value.

This happens because increasing the code size is not the only effect caused by procedure inlining. As mentioned before, inlining eliminates the instructions associated with the calling overhead. Aggressive inlining involves the removal of a higher number of these instructions, limiting the increase in the number of instruction cache misses. Moreover, inlining removes procedure boundaries, increasing the visibility of the code to other optimizations, like dead code elimination or code scheduling, potentially reducing the number of instruction cache misses. Nevertheless, this only happens for intermediate values of the MRCS parameter, that is, when the impact on the cache miss rate is not too high. The higher values of MRCS always cause a higher number of instruction cache misses.

To summarize, aggressive inlining involves an increase in the length of instruction streams, increasing the ability of the next stream predictor of tolerating the prediction table access latency, and thus improving the processor performance. On the other hand, aggressive inlining causes an increase in the number of instruction cache misses, which degrades the processor performance. In order to find the optimal setup, we explore this tradeoff in the next section.

5 Performance Evaluation

Both the length of instruction streams and the number of instruction cache misses have an important impact on the overall processor performance. In this section, we evaluate the processor performance looking for a balance between the average stream length and the instruction cache miss rate. We provide data for two processor setups, a 4-wide processor and an 8-wide processor. The wider processor requires longer streams to keep the execution engine busy while a new prediction is being generated, limiting the benefits of an aggressive inlining. Finally, to achieve a better understanding of our results, we provide data for each individual benchmark.

5.1 4-Wide Processor Performance

Figure 5 shows the 4-wide processor performance using a realistic prediction table access latency (3 cycles) without prediction overriding. We vary the MRCS value from 256 to 16384 instructions, and the total instruction cache hardware budget from 8KB to 64KB. The bigger the cache is, the more aggressive is the inlining that can be performed. Thus, the optimal value of MRCS is higher for the bigger cache sizes. Both the 8KB and 16KB instruction caches achieve their optimal performance using a 1024-instruction MRCS value, while the 32KB and the 64KB instruction caches achieve their optimal performance using a more aggressive 8192-instruction MRCS value.

The best performance is achieved by the 64KB instruction cache due to its lower miss rate. Using this cache, the code inlined using the optimal MRCS value achieves a 7% performance improvement over the non-inlined code. However, this improvement is not necessarily caused by the ability of tolerating the predictor access latency. It can also be caused by the higher fetch bandwidth provided by longer streams and the additional code optimizations enabled by our aggressive inlining. In order to provide more insight about this, we have measured the performance achieved by a processor with a 1-cycle latency predictor, where overriding has no impact on performance.

With this ideal latency predictor, the code inlined using the optimal MRCS value achieves a 6% performance improvement over the non-inlined code. Since the predictor access latency is not a problem for the ideal latency predictor, this improvement can be attributed not to the ability of tolerating the access delay, but to the higher fetch bandwidth and the additional code optimizations enabled by inlining. However, the performance improvement achieved by the realistic latency predictor is higher. Therefore, this additional 1% performance improvement is caused not only by the improved fetch bandwidth and the additional code optimizations, but also by the ability of longer streams of hiding the prediction table access latency.

Figure 6 shows the performance achieved by the 4-wide processor using the optimal MRCS value for each instruction cache size. As mentioned before, these results correspond to a realistic prediction table delay without overriding. This data is compared against the performance

IEEE Computer Society

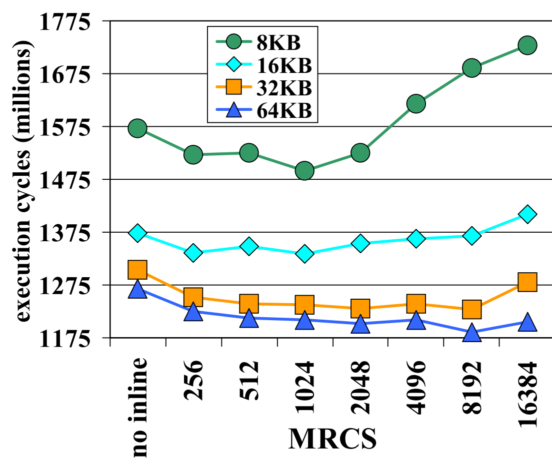


Figure 5. Average performance of a 4-wide processor varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

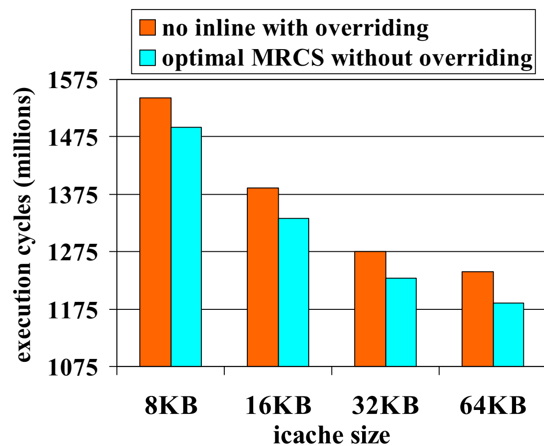


Figure 6. Average performance of a 4-wide processor with prediction overriding not using inlining, and without prediction overriding but using inlining and the optimal MRCS value.

achieved by a code optimized without inlining but using a hardware prediction overriding mechanism. The main observation is that, when using aggressive procedure inlining, a 4-wide processor using the stream fetch engine without overriding is able to outperform a similar processor using code optimized without inlining, even if it uses prediction overriding. Using a 64KB instruction cache, the processor without overriding executing inlined code achieves a 5% reduction in the total number of execution cycles over the processor executing non-inlined code using overriding. This illustrates how the fetch engine complexity can be reduced by exploiting the advantages provided by code optimizations.

5.2 8-Wide Processor Performance

A wider processor increases the pressure over the fetch architecture. Instructions are read from the fetch engine at a higher rate because the processor is able to execute a higher number of instructions per cycle. This fact limits the benefits achievable by using long streams of instructions, and thus the benefits obtained using aggressive procedure inlining. Therefore, an 8-wide processor requires either longer instructions streams than a 4-wide processor, or new stream predictions more often.

Figure 7 shows the 8-wide processor performance using a realistic 3-cycle prediction table access delay without prediction overriding. The MRCS value is varied from 256 to 16384 instructions, and the instruction cache is varied from 8KB to 64KB. The optimal MRCS values are the same found for the 4-wide processor setup, that is, 1024 instructions for the 8KB and 16KB caches, and 8192 instructions for the 32KB and 64KB caches. This means that the length of instruction streams is the same for both processor setups. Since the higher number of instructions per cycle needed by the 8-wide processor cannot be obtained from longer streams, it requires new predictions more often. This involves that the ability of tolerating the prediction table access latency is reduced.

Therefore, obtaining longer streams is more necessary for this processor setup than for the 4-wide setup. Once again, we have measured the performance achieved by a processor with a 1-cycle latency predictor in order to distinguish between the benefits of tolerating the access latency and the other factors that improve performance, i.e. the higher fetch bandwidth and the additional optimizations enabled by our aggressive inlining. Using a 64KB instruction cache and the ideal latency predictor, inlining provides a 6% performance improvement. When the predictor access latency is taken into account, that is, using the realistic 3-cycle latency predictor, the improvement rises up to 8%. This additional 2% improvement, caused by the ability of tolerating the access latency, is higher than the one obtained in the 4-wide setup. This fact highlights the importance of having long prediction units in wide processors.

Figure 8 shows the performance achieved by the 8-wide processor using a realistic predictor delay without overriding. Each instruction cache size is evaluated using the optimal MRCS value. This data is compared against the performance achieved by a code optimized without inlining but using a hardware prediction overriding mechanism. In spite of the higher fetch bandwidth requirements of the 8-wide processor, the stream fetch engine without overriding is still able to outperform a similar processor using code optimized without inlining, even if it uses prediction overriding.

However, the improvement is lower than in the 4-wide setup. Using a 64KB instruction cache, the processor without overriding executing inlined code only achieves a

0-7695-2061-8/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

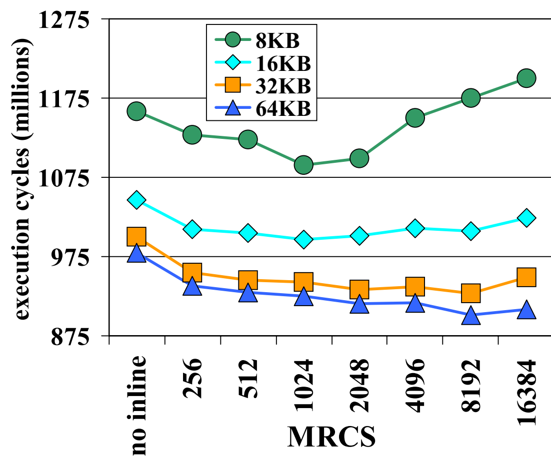


Figure 7. Average performance of an 8-wide processor varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

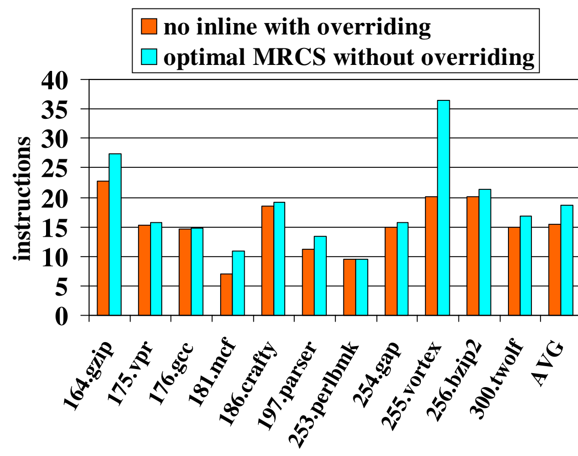


Figure 9. Stream length for individual benchmarks using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

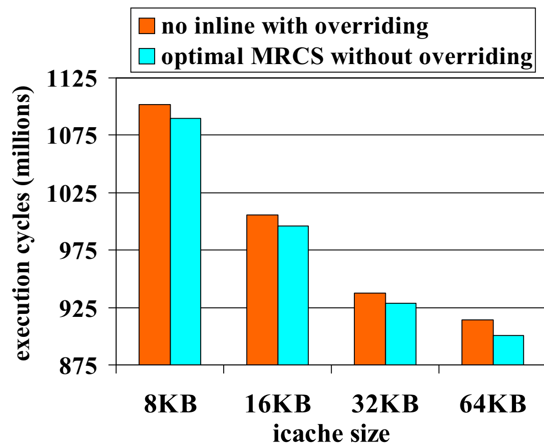


Figure 8. Average performance of an 8-wide processor with prediction overriding not using inlining, and without prediction overriding but using inlining and the optimal MRCS value.

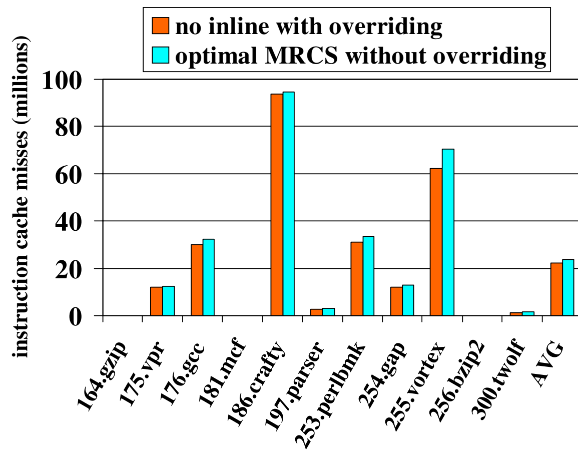


Figure 10. Instruction cache misses for individual benchmarks using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

1.5% reduction in the total number of execution cycles over the processor executing non-inlined code using overriding. This reduction in the benefit of aggressive procedure inlining shows that there is still room for improvement. If we can obtain longer streams, maintaining the instruction cache miss rate under control, the execution engine of wide superscalar processors can be feed during multiple cycles by a single stream, hiding the latency of the next stream prediction. Thus, longer streams will allow wide processors to achieve performance improvement results as good as the achieved by the 4-wide processor or even better.

5.3 Individual Benchmark Performance

To achieve a better understanding of our results, we show individual benchmark data for the 8-wide processor setup using an 8KB instruction cache with the optimal MRCS. We selected this cache because its small size aggravates the problem of instruction cache misses. Figure 9 shows the average length of instruction streams for both the code optimized without inlining and with inlining. There is a clear stream enlargement in six of the eleven evaluated benchmarks, specially in *255.vortex*, while the other five benchmarks present little variation. This enlargement is due to an

IEEE COMPUTER SOCIETY

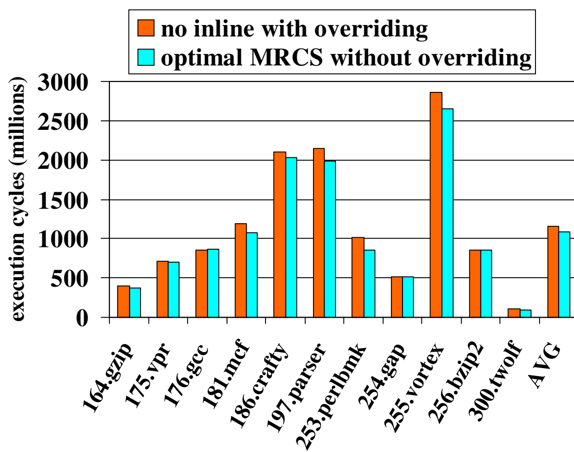


Figure 11. Processor performance for individual benchmarks in an 8-wide processor using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

important reduction in the number of taken branches. The six benchmarks that obtain longer streams using inlining achieve a reduction over 20% in the total number of function calls and return instructions, reaching a reduction over 90% in benchmarks like *164.gzip*. Both procedure calls and returns are specially frequent in *255.vortex*, which explains the high increase this benchmark achieves.

As stated before, this enlargement of instruction streams is obtained at the cost of a higher number of instruction cache misses. Figure 10 shows the number of instruction cache misses for the 8KB instruction cache using both the code optimized without inlining and with inlining. Inlining causes an increase in the number of instruction cache misses for *176.gcc*, *253.perlbmk*, and, specially, *255.vortex*. The increase in *255.vortex* is expected due to the great increase in the length of instruction streams. This is the price that must be paid for obtaining so long streams.

Figure 11 shows the performance achieved by the eleven benchmarks in the 8-wide processor using the 8KB instruction cache setup, and both the code optimized without inlining using prediction overriding, and the code optimized with inlining not using overriding. The performance of both processors is close in most benchmarks. The processor without overriding, in spite of being less complex, is even better in five benchmarks. A clear example is *255.vortex*, which obtains an important reduction in the execution time using aggressive procedure inlining. Although it suffers from a higher number of instruction cache misses, the longer streams obtained in *255.vortex* when using inlining compensate this effect, allowing to achieve a better performance by hiding the prediction table access latency. This shows that software optimization techniques are able to replace a complex hardware mechanism, like prediction overriding, reducing the fetch engine cost and complexity without sacrificing performance.

6 Conclusions

Current technology trends create new challenges for the fetch architecture design. Higher clock frequencies and larger wire delays cause branch prediction tables to require multiple cycles to be accessed [1, 8], limiting the fetch engine performance. This has lead to the development of complex hardware mechanisms, like prediction overriding [8, 19], to hide the prediction table access delay.

In order to avoid this increase in the fetch engine complexity, we propose to use a software approach to hide the prediction table access delay. The next stream predictor [14] is a branch predictor specially designed to take advantage of code optimizations. It uses instruction streams as basic prediction unit. We call stream to a sequence of instructions from the target of a taken branch to the next taken branch. If instruction streams are long enough, the execution engine can be kept busy executing instructions from a stream during multiple cycles, while a new stream prediction is being generated. Therefore, the prediction table access delay can be hidden without requiring any additional hardware mechanism.

Our previous work [14] shows that profile-directed code reordering is able to enlarge instruction streams. In this paper, we use aggressive procedure inlining for obtaining even longer instruction streams. Inlining is a commonly used code optimization that replaces a procedure call by the procedure itself. Procedure inlining enlarges instruction streams by reducing the number of function calls and return instructions, that is, reducing the number of taken branches. This enlargement of streams involves that each prediction generated by the stream predictor contains a higher number of instructions, increasing its ability for tolerating the prediction table access latency.

However, aggressive inlining also involves an increase in the number of instruction cache misses due to the code increase. Therefore, it is important to correctly balance the stream length and the instruction cache miss rate in order to achieve the best performance. We have explored different levels of inlining aggressivity, showing the optimal one for different instruction cache sizes. As can be expected, larger instruction caches tolerate more aggressive procedure inlining, allowing to obtain longer streams. In general, procedure inlining provides streams long enough for allowing a processor not using overriding to outperform a similar processor executing code optimized without inlining, even if it does use prediction overriding. These results show that taking into account the interaction between software code optimizations and the fetch architecture, it is possible to achieve a high performance at a low cost and complexity.

Acknowledgements

This research has been supported by CICYT grant TIC-2001-0995-C02-01, CEPBA, and an Intel scholarship grant. O. J. Santana is also supported by Generalitat de Catalunya grant 2001FI-00724-APTIND. In addition, we would like to thank Manel Fernández and David Kaeli for their worthwhile help with the ALTO tool.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[2] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. *Proceedings of the Conference on Programming Language Design and Implementation*, 1988.

[3] H. Aydin and D. Kaeli. Using cache line coloring to perform aggressive procedure inlining. *ACM Computer Architecture News, vol. 28, no. 1*, 2000.

[4] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. *Proceedings of the Conference on Programming Language Design and Implementation*, 1997.

[5] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[6] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6-8 fo4. *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[7] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the Conference on Programming Language Design and Implementation*, 1989.

[8] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[9] A. KleinOsowski and D. J. Lilja MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters, vol. 1*, 2002

[10] R. Muth, S. K. Debray, S. A. Watterson, and K. De Bosschere. alto: a link-time optimizer for the Compaq Alpha. *Software - Practice and Experience, vol. 31, no. 1*, 2001.

[11] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[12] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[13] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. *U.S. Patent Number 5,381,533*, 1995.

[14] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[15] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[16] E. Rotenberg, S. Benett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[17] O. J. Santana, A. Ramirez, J. L. Larriba-Pey, and M. Valero. Accurate Latency-Tolerant Branch Prediction. *Technical Report UPC-DAC-2003-09*, 2003.

[18] O. J. Santana, A. Ramirez, and M. Valero. Latency Tolerant Branch Predictors. *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2003.

[19] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[20] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. *Western Research Laboratory Research Report 2001/2*, 2001.

0-7695-2061-8/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY