EE 482A Project
Spring 2000

Application of Zero Overhead Loop and Address Generator in General Purpose Computing

Jinyung Namkoong

1. Introduction

Zero overhead loop (ZOL) support for efficient 'for' loop implementation and dedicated address generator (AGEN) approach for prefetching a stream of data are two of the techniques that are common in DSP processors. While going through some of the papers in this class, it seemed to me that much of the problems of branch prediction and (data) memory latency could be solved by these same DSP techniques, which seem to have lower hardware complexity. In this project, I have investigated how these DSP techniques could be applied to general purpose computing, and how they would fare against more common techniques we have encountered in this class.

Since these techniques are specifically targeted for DSP applications, where both the instruction and data access patterns are quite regular and much more predictable, it was expected from the beginning that they would not attain their full utility in general purpose benchmark such as SPEC95. Thus the question that's addressed in this project is exactly to what extent ZOL and AGEN approaches can be applied to the SPEC95 benchmarks. This coverage is evaluated by running precompiled SPEC95 benchmark applications on SimpleScalar processor model and extracting relevant profile information. Extremely simplified performance models are developed to evaluate how these DSP techniques would fare depending on some processor parameters. As it turns out, examination of the precompiled code uncovered very few places where ZOL and AGEN can be applied directly. This is partly due to the fact that these general purpose benchmarks inherently exhibit much more irregular access patterns, but the fact that I looked only at precompiled code may be equally as important. If these techniques were to be actually employed, compiler's responsibility for utilizing these features would be all-important. Thus, examining only the assembly code compiled for conventional machine is of questionable validity, other than the fact that it's so much simpler to do given the fast approaching deadline and panicked state of mind. That confession aside, the profile information still shows some interesting patterns of typical dynamic execution, and these are explained in later sections.

This report is organized as follows. In section 2, possible incorporations of ZOL and AGEN into general purpose CPU, assumed while designing and performing experiments, are shown, as well as what problems they can potentially solve. In section 3, the experimental setup to evaluate how much these techniques can be applied is explained. Section 4 is the hypothesis, Section 5 is the experimental results and analysis, and Section 6 is conclusion.

2. ZOL and AGEN applications

2.1 Zero Overhead Loop (ZOL)

ZOL is an efficient mechanism for implementing 'for' loops. Before entering the loop segment, some registers specifying the number of times the loop is to be run (L) and the beginning (START_PC) and end (END_PC) of the loop body are set up. Then, without explicit branches embedded in the code, the loop body is executed the L times before proceeding to the next segment. Typically there is a loop counter which is initialized to L and is decremented with each execution of the loop. The 'zero overhead' comes from the fact that the branch instruction and another instruction used to increment/decrement the counter are eliminated, resulting in fewer number of dynamic instructions. In this respect, it can be thought of as hardware loop unrolling, without the increased code size associated with software loop unrolling. (The term 'hardware loop unrolling' can be misleading, since it is still the compiler which makes decision to use this feature, except that it uses the hardware mechanism to achieve loop unrolling.)

There can be different implementations of ZOL, but the basic concept is the same. For example in DSP16000 [1], there are a loop counter, a register containing the number of instructions in the loop, a pointer to the next instruction in the loop, and a loop buffer containing all the instructions in the loop. This loop buffer, which functions as software managed cache in a typically cache-less DSP, is not necessary for our purpose, since the instruction cache will already be present. Whereas using I-cache as loop buffer creates the possibility of the buffer being corrupted, it also allows for multiple zero overhead loops to be active at the same time. In this case, there will be a register file of ZOL status registers (loop_begin, loop_end, loop_cntr, valid, etc). Each cycle, every loop_end is compared against the current PC, and when a valid active loop entry is found, the next PC is changed to loop_begin, while loop_cntr is decremented by 1. ZOL is conventionally used only for usually taken, backward branches. However, this implementation with multiple ZOL entries is more general, and can be used even in place of forward branches, if that branch is known to be taken a certain number of times. It could even replace usually not taken branches, if such is found to be useful.

In this project, ZOL is not actually implemented into simulator. Instead, certain profiling information of dynamic execution of SPEC95 applications is used to estimate ZOL's potential application and coverage, as will be explained in the next section. ZOL approach was initially chosen as a competitor for hardware branch prediction, since ZOL effectively converts conditional branches into deterministic ones. However, it became apparent early on that ZOL coverage is not very significant, and to be of any usefulness, it will have to be combined with static branch prediction scheme. So, the comparison now becomes dynamic branch prediction vs. static branch prediction with ZOL. To compare only the effect of branch prediction and ZOL, an idealized model is used. In this model, all other sources of single thread

application bottleneck (data dependency, memory latency, structural hazards, branch target calculation, etc) are ignored, and only the issue width and the branch miss penalty are modeled. Then, the equation for # of cycles becomes:

$T = \#\ of\ cycles\ to\ run\ a\ program$

$N = \#\ of\ dynamic\ instructions$

$W = Issue\ Width$

$P_D = Branch\ Miss\ Rate\ for\ Dynamic\ Branch\ \mathrm{Pr}ediction$

$P_S = Branch\ Miss\ Rate\ for\ Static\ Branch\ \mathrm{Pr}ediction$

$BBsize = average\ basic\ block\ size$

$X = Branch\ Miss\ Penalty\ in\ \#\ of\ Cycles$

$d = fraction\ of\ dynamic\ instructions\ that\ are\ removed\ by\ ZOL$

$$T_{Dynamic} = N\left(\frac{1}{W} + \frac{P_D}{BBsize}\left(X + \frac{1}{2}\right)\right)$$

$$T_{Static+ZOL} = N\left(\frac{1-d}{W} + \frac{P_S\left(1 - \frac{d}{2}BBsize\right)}{BBsize}\left(X + \frac{1}{2}\right)\right)$$

It is assumed that for each iteration of the loop that can be converted to ZOL, two dynamic instructions are removed as overhead, one for the branch and another for keeping track of loop index. This would be true if the loop index in 'for' loop is used only for counting the number of iterations and nothing else. ½ factor comes from the fact that depending on the place where the branch misprediction occurs, an average of ½ of the W issue slots would be wasted, in addition to X branch miss penalty proper.

Though this is an over-simplified model that has no resemblance to actual performance measure, it shows some trends that may be relevant. First, it is well known that P_S is generally lower than P_D. The question is how much can be regained by addition of ZOL. It is apparent that ZOL helps in two ways: by eliminating loop overhead instructions, and by converting a number of branches that might otherwise cause branch misprediction. These two are related because the number of overhead instructions removed is simply two times the number of branches removed. The equation also confirms our intuition that when issue width W is smaller, the code size reduction plays more important role, and as W gets larger, lower misprediction rate becomes more dominant.

I expect that most other factors not modeled by this simple equation will probably affect the two approaches similarly, thus not changing the basic relative

characteristic. Thus, they'll only act to decrease the relative performance gap between these two approaches.

2.2 Address Generator (AGEN)

Address Generators are used to fetch or write back streams of data, when the data access pattern is regular. If this were to be incorporated into the general purpose CPU, there will be several address generators, each with its own stream buffers. Compiler detects when the data access pattern is regular and AGEN can be utilized, and issues an AGEN instruction with proper access pattern. Then, AGEN keeps its associated stream buffer full, internally generating addresses based on the previous AGEN instruction. There will be special load (and possibly store) instruction, which gets its data not from the cache but from the stream buffer.

AGEN is useful in two ways. First, by bypassing the cache altogether, the data availability is guaranteed, not subject to cache behavior. Second, because it does not use cache, the cache is reserved for more random data access, which it handles the best. If all the data accesses are deterministic, cache is less crucial. Cache attempts to capture otherwise random accesses by spatial and temporal locality, and it is really an overkill for deterministic accesses like streams. Since I expect these streams to be quite long, it could potentially wipe out useful active data sets. Therefore, by redirecting these deterministic streams out of the cache, the whole program benefits.

That being said, I realize that address generator concept is nothing new even in general purpose computing. In a typical VLIW machine, there is a I-unit whose primary function is just generating addresses. The difference here is the proposal to dedicate one separate FIFO buffer through which deterministic traffic can travel.

In this project, I analyze the memory access pattern to detect such stream-like patterns, and see what kind of effect it has on the number of cache misses.

3. Experimental Setup

For both ZOL and AGEN evaluations, the SimpleScalar's functional simulators are used to gather profile information. All simulation results are based on 100 million dynamic instructions, after the initial 500 million instructions.

3.1 Measuring ZOL coverage

SimpleScalar's sim-safe functional simulator is used to gather dynamic branch behavior of SPEC95 benchmarks. This information is used to calculate static misprediction rate, dynamic misprediction rate, and the number of dynamic branches that can be converted to ZOL.

There are three flavors of static branch prediction, depending on how to decide whether a particular branch is biased taken or not taken. The three different static schemes are GlobStatic, AppStatic, and InstrStatic. In the first two schemes, a separate decision is made for all combination of branch instruction type and branch direction. For example, {beq, forward}= T, {beq,backward}= NT, {bne,forward}=NT, {bne,backward}=NT, etc. In GlobStatic, each decision is based on composite profile of all SPEC95 benchmarks, and the same assignment is used for all different benchmarks. In AppStatic, each benchmark is profiled and its decision made separately. Thus, AppStatic is always better than or equal to GlobStatic. For InstrStatic, each of the branch instructions is profiled separately, yielding even better prediction performance.

There are many possible implementation of dynamic predictors, but for the purpose of this project, Gshare with equal number of global history bits and branch PC is used. For the pattern history table size of N entries, $\log_2(N)$ bits of global history and $\log_2(N)$ bits of lower PC bits (excluding the least significant 2 bits, which is always 00) are simply XORed together to get access pointer to the pattern history table.

To see how many of dynamic branch instructions can be converted to ZOL, the branch distance for each of the branch instruction is observed to see whether there is a definite loop behavior. Branch distance is defined to measure how far the current condition is from the T-NT transition point. For example, for beq instruction, the branch distance is defined to be RS-RT, and for bgtz, it is defined to be RS. If this distance changes by a certain stepsize each time this branch instruction is executed, followed by a sudden jump, this branch is assumed to be convertible to ZOL. For example, a particular branch instruction with 4,3,2,1,0,4,3,2,1,0,… distance pattern is considered ZOL convertible, whereas another one with 4,3,2,3,5,1,0,… distance pattern is considered not ZOL convertible. In fact, since convertible branch will be eliminated from the instruction altogether, the distance pattern has to be completely predictable for the branch instruction to be considered ZOL convertible. That means that 4,3,2,1,0,0,4,3,2,1,0,0, is not acceptable, because of repeating 0. However,

different stepsize for different iteration is acceptable, such as 4,3,2,1,0,6,4,2,0,9,6,0,etc.

3.2 Measuring AGEN coverage

AGEN coverage is measured in similar way to ZOL coverage. Basically, the data access pattern for each static load or store instruction is examined to see whether it exhibits stream-like behavior. If so, this instruction is tagged as AGEN convertible, and cache is simulated without this instruction requesting to use the cache.

First, SimpleScalar's sim-cache functional simulator is used to measure the number of data cache accesses and the miss count. Then, sim-safe is used to detect regular pattern and single out the instructions that can use AGEN. Then, sim-cache is run again, except that now the instructions marked as AGEN convertible do not access the cache. The number of misses is compared against the first run.

Identifying certain data accesses as streams and moving them out of the cache is bound to decrease the number of cache misses by definition. Since the accesses to stream buffers are assumed to be hit all the time, the overall memory stall cycles are reduced. However, the big assumption here is that the data elements that are accessed as stream are not accessed in random fashion, at least not at the same time. Since the whole point of adding AGEN support is to bypass the cache, if the same set of data is accessed both randomly and also deterministically, there's no point of using AGEN, since the data would already be in the cache. In the experimental result that follows, this effect is not considered. The experiments simply identify certain memory instructions that access the data in regular fashion, and remove them for the second cache simulation. Thus, it is not very representative of the actual program behavior. However, the purpose of this project is simply to see how much regularity there is in data access pattern, for which AGEN can potentially help, rather than exactly simulating various effects of the implementation. Cache part is there mostly for fun only.

4. Hypothesis

I expected to see a lot of regularity both in branch distance pattern for each branch instruction, and also the data access pattern for each load or store instructions. Though it was expected that to reap the maximum benefit of ZOL and AGEN, compiler has to be targeted for these features, I still expected at least 20~30% of the dynamic branches and memory operations to be convertible to ZOL and AGEN, respectively. Also, I expected to see a lot more regularity in fp applications, thus more coverage of ZOL and AGEN in these benchmarks. The rationale was that since these scientific applications would deal a lot with matrices, and since the access pattern for these matrices would be regular and a lot of 'for' loop would be used to handle them, these must have over 50% coverage. (By coverage, I mean how many of the dynamic branches and memory ops would be convertible to ZOL and AGEN.)

Though more realistic performance impact of these techniques is not analyzed, it was expected that if the coverage is high enough, these techniques could actually compare favorably against more expensive alternatives. In case of ZOL, this could make simpler branch prediction viable, and in case of AGEN, smaller or simpler cache could be used. The cost of implementing these techniques is assumed minimal. The compiler would have to be smarter to identify regular patterns and apply these features, if these were to be adopted for real, but the added complexity is not expected to be high.

5. Experimental Results and Analysis

In this section, the simulation results are explained.

5.1 ZOL

The simulation results for ZOL are summarized below:

	go	compress	m88ksim	perl	hydro2d	su2cor	tomcatv
BBsize							
	6.77	5.09	6.24	5.21	7.99	4.2	4.69
static mispred.							
GlobStatic	32.62%	16.55%	24.60%	24.13%	35.17%	22.47%	19.54%
AppStatic	32.62%	16.55%	15.21%	24.13%	19.63%	21.06%	17.94%
InstrStatic	17.72%	16.55%	3.73%	6.43%	2.92%	2.61%	2.16%
dynamic mispred.							
1k PHT	29.24%	14.10%	6.81%	13.41%	0.09%	6.42%	8.38%
4k PHT	24.76%	12.90%	3.88%	6.68%	0.09%	1.87%	3.41%
16k PHT	19.31%	10.53%	2.76%	3.10%	0.09%	0.20%	1.65%
ZOL coverage							
% branch reduction	0.09%	3.70%	14.41%	0.26%	0.00%	3.21%	2.59%
% instr. reduction	0.03%	1.45%	4.62%	0.10%	0.00%	1.53%	1.11%

As expected, instrStatic performs noticeably better than the other two static schemes, exceeding even the dynamic predictors with small table size. However, the performance of dynamic predictors improves with increasing table size and easily surpasses InstrStatic in all benchmarks. However, as shown in the simplified performance equation of Section 2, this inadequacy of static predictor could potentially be offset by ZOL, if sufficient number of dynamic branches could be converted to ZOL. However, as shown in the table, this coverage was surprisingly small in most of the benchmarks, except in m88ksim. Contrary to the initial expectation that ZOL would be widely applicable to fp benchmarks, where large data sets are handled, hopefully in ‘for’ loop like structures, fp benchmark coverage was not much different from int benchmark coverage. From examining the branch traces, it was found that most of the branch distance turns out to be either 0 or 1, as in 0,0,0,0,1,1,0,0,1,0,… distance pattern. Also, some of the distance patterns exhibit some step-like behavior, but are not exactly ZOL convertible, like in 7,6,5,4,3,3,7,6,5,4,3,2,1,… etc. As mentioned in Section 1, this low coverage result may very well be because I’m starting at a wrong level. Much more coverage may be obtainable, if the problem was solved from the compiler level.

The simplified performance equations of Section 2 are duplicated below:

$$T_{Dynamic} = N\left(\frac{1}{W} + \frac{P_D}{BBsize}\left(X + \frac{1}{2}\right)\right)$$

$$T_{Static+ZOL} = N\left(\frac{1-d}{W} + \frac{P_S\left(1-\frac{d}{2}BBsize\right)}{BBsize}\left(X + \frac{1}{2}\right)\right)$$

Using the data on the table above, these equations can be evaluated. The m88ksim (best coverage case) with 16K PHT as dynamic predictor and AppStatic as static predictor is taken as an example. Then, the equation becomes:

$$CPI_{Dynamic} = \frac{T_{Dynamic}}{N} = \frac{1}{W} + \frac{0.0276}{6.24}\left(X + \frac{1}{2}\right) \cong \frac{1}{W} + \frac{1}{226}\left(X + \frac{1}{2}\right)$$

$$CPI_{Static+ZOL} = \frac{T_{Static+ZOL}}{N} = \frac{1-0.0462}{W} + \frac{0.1521(0.8558)}{6.24}\left(X + \frac{1}{2}\right) \cong \frac{0.9538}{W} + \frac{1}{48}\left(X + \frac{1}{2}\right)$$

Then, we can see how they compare as W and X are varied: (m88ksim)

Dynamic	W=1	W=2	W=3	W=4
X=1	1.007	**0.507**	**0.340**	**0.257**
X=2	1.011	**0.511**	**0.344**	**0.261**
X=3	**1.015**	**0.515**	**0.349**	**0.265**
X=4	**1.020**	**0.520**	**0.353**	**0.270**

Static+ZOL	W=1	W=2	W=3	W=4
X=1	**0.985**	0.508	0.349	0.270
X=2	**1.006**	0.529	0.370	0.291
X=3	1.027	0.550	0.391	0.311
X=4	1.048	0.571	0.412	0.332

Thus, even for the relatively high coverage of m88ksim (δ = 4.62%), Static+ZOL performs better than Dynamic only for low two entries shown above. As evident from the equation itself, Dynamic performs better for the lower right hand corner (high W and X,) and Static+ZOL performs better for the upper left hand corner (low W and X.)

5.2 AGEN

The simulation results are summarized below:

	go	compress	m88ksim	perl	hydro2d	su2cor	tomcatv
a) total number of dynamic memory ops (loads + stores)							
total	29,320,364	9,452,616	36,725,403	45,214,121	23,513,605	31,568,701	29,839,654
b) number of dynamic memory ops that accesses the same location as previous access							
zerostep	11,386,978	2,911,517	27,634,315	17,669,009	4,196,004	25,455,699	22,283,676
	38.84%	30.80%	75.25%	39.08%	17.85%	80.64%	74.68%
c) total number of streams identified (minimum stream length = 16)							
tot#str	44	1	2	8	299	4	2
tot#els	9057	727124	64	612	7367192	192513	52574
%	**0.03%**	**7.69%**	**0.00%**	**0.00%**	**31.33%**	**0.61%**	**0.18%**
d) those that belong to stream-only memory ops							
#strOps	27	1	0	0	113	2	1
#streams	27	1	0	0	113	2	1
#elements	7324	727124	0	0	4284621	190984	52574
	0.02%	**7.69%**	**0.00%**	**0.00%**	**18.22%**	**0.60%**	**0.18%**
				pwd			
e) Original # of L1 cache misses							
#misses	3335198	2591471	546541	2741335	3562408	414456	496035
	11.38%	27.42%	1.49%	6.06%	**15.15%**	1.31%	1.66%
f) # of L1 cache misses with stream ops removed							
#misses	3333902	2525780	546541	2741335	2681814	414456	496035
	11.37%	26.72%	1.49%	6.06%	**11.41%**	1.31%	1.66%

First thing to notice (part b) is that there are quite significant number of loads and stores that keep accessing the same memory locations repeatedly. This looks weird, since if these values are so frequently needed, it makes sense to keep them in internal registers rather than going out to memory every time. In one extreme example of compress benchmark, there are only 7 static load and store instructions that get executed over 9 million times. Of those, 4 static instructions, corresponding to close to 3 million memory ops, access only one memory location per instruction. If there were four spare registers, these memory ops could have easily been removed. This weird effect may be due to lack of compiler optimizations. I initially mistook zero stride accesses for constant stride accesses, which are AGEN convertible, which gave overly optimistic expectation.

Part c) of the table shows the total number of streams identified. tot#str shows the total number of distinct streams, and tot#els means total number of data words within the streams. tot#els is the number of dynamic memory ops that could be redirected to stream buffers, and the percentage shows the ratio of this tot#els to total # of memory ops. A stream is identified if more than 16 accesses are made with constant stride.

In part c), all streams are counted, regardless of whether these streams comes from the static instructions that only have regular access pattern, or those that also have random access pattern. If these memory instructions are to be converted to different type of instructions that go through AGEN, only those streams coming from 'pure' stream memory instructions should be counted. Part d) is just that, thus a subset of part d). As shown, in all of these benchmarks, there seems to be one stream per a static instruction, which may simply be because of short simulation period.

Because the coverage is so small, the cache simulation comparison is almost meaningless for most applications. The cache simulated is direct-mapped 8KB L1 data cache with 32B block size. Part e) lists numbers of misses without AGEN, and part f) lists numbers of misses with AGEN. Except for hydro2d, where AGEN coverage is significant, there's virtually no difference in the number of cache misses. In fact, for some applications like su2cor, the removed data accesses must all be the ones already in the cache, since the number of cache misses doesn't change at all.

Although the portion of data access that can be redirected to AGEN/stream buffer is not large, this experiment has shown that there are indeed data access patterns that fit into AGEN/stream buffer approach perfectly, and with compiler optimizations, it has potential to find more coverage.

6. Conclusion

The goal of this project was to investigate to what extent ZOL and AGEN, two of the common DSP techniques, could be applied to general purpose computing. The themes for both these techniques were similar: if there is identifiable regular pattern in branch distance (how far away from the transition condition) or in data access history, the corresponding conditional branch may be converted to deterministic ZOL, and the corresponding memory op may use AGEN and associated stream buffer, without going to cache and potentially corrupting it unnecessarily.

ZOL with static branch prediction is compared against dynamic branch prediction scheme. Though the misprediction rate for static scheme is known to be higher than that for the dynamic scheme, the purpose was to see whether the code and branch reduction by ZOL would offset the effect of higher misprediction rate. The coverage of ZOL was measured by examining dynamic branch behavior, and detecting predictable behavior that may be converted to ZOL. As it turns out, the coverage was not high enough for ZOL+Static to surpass Dynamic in most cases. Surprisingly, the floating point coverage was not any greater than integer benchmark coverage. There were a lot of branch distance patterns comprising only of 0 and 1, not useful for ZOL.

AGEN is proposed as an extension of the stream buffer/prefetching scheme. Here, the purpose was to identify memory ops whose access pattern can be converted to streams, and to simulate cache with and without these memory ops. However, contrary to expectation, the number of streams detected was not large in most of the applications. It was found that there is a very large fraction of 'zero-stepsize-stream,' the kind that's not convertible to AGEN/stream buffer.

These DSP techniques turned out not to be as widely applicable to the general purpose benchmarks as initially anticipated. ZOL and AGEN approaches can be very efficient when the data and instruction access patterns are regular and easily predictable. However, the access patterns for general purpose applications are a lot more random, and in these applications, the utility of ZOL and AGEN is limited. It may be possible to increase the utility of these techniques if compilers are targeted to use these features.

With increasing dominance of multimedia content in general purpose computing, and blurring of wall between DSP and general purpose CPU, adding these features may be a viable alternative to make general purpose CPU run DSP applications better, and to prevent large data sets of DSP applications from monopolizing valuable processor resources.

7. Reference

G.R.Uh, Y.Wang, D.Whalley,S.Jinturkar, C.Burns, V.Cao, “Effective Exploitation of a Zero Overhead Loop Buffer,” Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, May 5, 1999