Compiler-Directed Early Load-Address Generation

Ben-Chung Cheng* Daniel A. Connors† Wen-mei W. Hwu†
*Department of Computer Science
†Department of Electrical and Computer Engineering
The Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
Email: {bccheng, dconnors, hwu}@crhc.uiuc.edu

Abstract

Two orthogonal hardware techniques, table-based address prediction and early address calculation, for reducing the latency of load instructions have been recently proposed. The key idea behind both of these techniques is to speculatively perform loads early in the processor pipeline using predicted values for the loads' addresses. These techniques have required either a large hardware table or complex register bypass logic to be implemented in order to accurately predict the important loads in the presence of a large number of less-important loads. This paper proposes a compiler-directed approach that allows a streamlined version of both of these techniques to be effectively used together. The compiler provides directives to indicate which prediction mechanism to use or, when appropriate, that a prediction should not be made. The hardware therefore can be focused on their target cases so that a smaller prediction table and simpler bypass logic suffice. Our results show that through straightforward compiler heuristics, we obtain an average speedup of 34% with a 256-entry direct-mapped address table and only one cached register. And with the help of address profiling, an extra 4% of speedup can be obtained.

1 Introduction

test

The latency of load instructions can significantly affect overall performance of modern microprocessor architectures. Load instructions have longer latency than most computation instructions because they have both address calculation and memory access tasks to perform. As a result of the growing gap between main memory and processor clock speeds, the memory access latency has an even greater impact on performance, motivating the need for new load latency reduction techniques.

Various techniques for hiding the latency of load instructions have been investigated in the past. Reducing the load latency by accessing the memory earlier than the normal pipeline stage using early load-address generation has been addressed [1][2][3][4][5]. Similarly, the concept of value prediction [6][7] has introduced machine models that exceed the limit imposed by data dependences by predicting the outcome values of instructions. One major difference between these two types of techniques is that early load-address generation schemes may increase the overall memory bandwidth requirement due to speculative loads with wrong addresses. But for value prediction schemes, no speculative loads are needed since the value is predicted and verified by the normal load. On the other hand, for value prediction schemes to effectively hide the load latency, dependent instructions need to be speculatively issued based on the predicted value. Therefore recovery techniques definitely need to be incorporated in value prediction schemes. In early load-address generation schemes, as long as the validity of the speculatively loaded data can be verified before the normal memory stage in the pipeline, its latency can be hidden and no recovery is needed. A comprehensive study of previous work reveals that the effective address of a load can be obtained through two orthogonal categories: to predict it based on a table or to calculate it earlier in the pipeline. We also find that both methods have strengths and weaknesses for different scenarios. In general both methods, as previously proposed, also potentially waste resources by using a significant amount of hardware space to generalize a mechanism to handle its sub-optimal cases. Therefore it is desired to have both methods available and to use the compiler to select the method based on their target cases.

In this paper we present a compiler-guided early

load-address generation scheme to hide the latency of load instructions. Our approach involves both instruction set and microarchitecture changes and compiler support. At the microarchitecture level, we add the functionalities of performing both table-based prediction and early calculation for load-address generation. Selection between the dual paths is performed by the compiler using new instruction opcodes. For each original opcode, enough information is added to the instruction encoding to differentiate three cases. When the new opcode indicating that the address can be predicted is used, a hardware table is accessed to determine the current address from the previous address and a possible stride. If the predicted address is correct and obtained before the real one is computed, the load latency can be reduced. Another new instruction opcode is used for early calculation. The register value for early calculation is retrieved from a special one-entry cache which can be treated as a special addressing register. Using an isolated addressing register requires only a limited broadcast between the register file and early calculation mechanism, making the register caching design more straightforward in the implementation. Finally, an instruction opcode indicates the unpredictability of a load to prevent it from polluting the prediction table and the addressing register.

We present an instruction set architecture and microarchitecture pipeline design for supporting compiler-directed early load-address generation for load instructions. We evaluate the concept of early address generation in the context of both our compiler-directed scheme and previously proposed hardware-only schemes. Results indicate that our approach is more effective in reducing the average latency of load instructions, at potentially lower hardware costs. With our approach, an average speedup of 34% is achieved with a 256-entry direct-mapped address prediction table and one compiler-directed special addressing register without instruction predecode and register multicasting. This speedup is comparable to that generated by a hardware-only scheme with much larger table and complex register-caching mechanism. We also find that our approach is flexible in that we can adapt compiler analysis and profiling techniques to further improve overall performance by better classifying the loads. The results show that an extra 4% of average speedup can be obtained using the address profile information. We also evaluate the suitability of our approach for embedded processors. Such processors may benefit the most from our approach since the impact of instruction set modification is minimal while the constraints on space and power consumption are rigorous.

The rest of this paper is organized as follows: Section 2 reviews the rationale and related work. Section 3 presents the instruction set and microarchitecture design necessary for supporting dual early address generation mechanisms. Section 4 discusses some initial compiler heuristics used to support our approach. Section 5 presents the experimental results. Finally, conclusions are stated in Section 6.

2 Rationale and Related Work

2.1 Rationale

In this section we briefly present some intuitive rationale behind early address generation. Figure 1 shows data hazards either caused (*load-use*) or suffered (*address-use*) by load instructions. Figure 1a shows the load-use hazard where a load is followed by an immediate use. In this paper we assume a six-stage pipeline design with two decoding stages and one cache-access stage. Since the load takes one cycle to calculate the effective address and takes another cycle to access the data cache, even though the instruction that uses the loaded register is issued one cycle later, it still suffers a one-cycle stall for the data.

As shown in Figure 1b, if an early address generation scheme produces the correct address and successfully brings the data back in the highlighted stages, the one-cycle stall disappears. Some aggressive schemes can even bring back the data as early as in the ID2 stage so that the subsequent use instruction can be issued in the same cycle as the load [1]. Figure 1c shows a case targeted by the table-based address prediction scheme. In this example, `r1` is incremented by four in each iteration, and the effective address for the load can easily be predicted from its previous address plus a constant stride, which is four in this case. Figure 1d shows the code that typically executes in pointer-chasing loops. Since `r1` is filled from memory in each iteration, its content is hard to predict as are the subsequent loads' addresses. Thus the stride-based approaches do not work under this situation. But with early address calculation and when the up-to-date value of `r1` can be retrieved early in time, both `r1+0` and `r1+28` can be computed to access the cache speculatively in the highlighted stages thus their latencies are reduced. However, code in Figure 1c presents the worse-case scenario for the early address calculation scheme since it transposes the hazards on its destination register to its base and index registers. When the early calculation stages are entered, `r1`'s up-to-date value is not ready yet so the speculatively calculated result is incorrect.

2.2 Related Work

Austin and Sohi [1] presented an early address calculation scheme which predecodes instructions when the

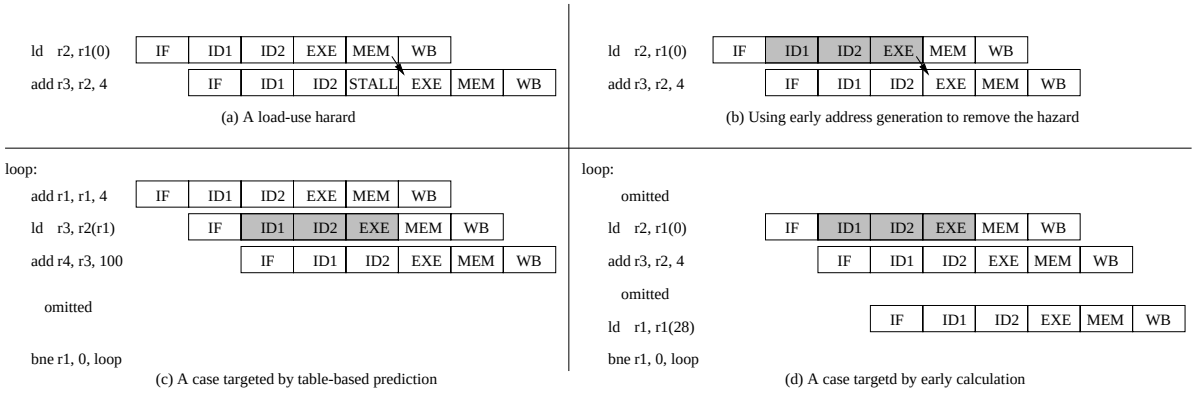


Figure 1. Various hazards involving load instructions.

instruction cache is filled. With the predecode information, a cache for base registers and index values (BRIC) is probed in the decode stage. When register files are updated, corresponding entries in the BRIC are updated as well. Depending on the format of the BRIC, multi-casting writes may be necessary. This approach is capable of providing the loaded data at the beginning of the execution (EXE) stage if the load uses register+offset addressing mode, or at the beginning of the memory (MEM) stage if the load uses register+register addressing mode.

Chen and Wu [2] proposed another early address calculation scheme with the stride-detection capability. In their scheme instruction predecode is avoided, but multi-casting writes are still required. However, since the estimated stride is added in the early address calculation path, it delays the address verification procedure. In this case, the maximum latency reduction is one cycle. Also, only latencies of loads using register+offset addressing mode can be reduced. In our approach strided addresses can be generated early in a more economical way without hurting the early address calculation path.

Several table-based address prediction schemes are presented in the literature. Golden and Mudge's scheme [4] employed a PC-indexed table to record the most recently used memory address for each load and store instruction. Sazeides, Vassiliadis and Smith's approach [8] is similar in that it can detect loads from constant addresses. Gonzalez [5] added saturating counters to prevent predictions for unpredictable loads after repeated incorrect predictions were made. All of these address prediction methods required a large hardware table to prevent the contention of table entries. These schemes are also unable to reduce the latency of loads whose addresses are not linear, such as pointer-chasing loads.

Eickemeyer and Vassiliadis' mechanism [3] also provides dual paths for early address generation. When load instructions are found in the instruction buffer, the load unit selects one of the address generation paths based on the run-time conditions. One major difference between our approach and their approach is that our method initiates early addresses during an instruction's pipeline execution. The alternative of initiating early addresses by processing the contents found in the instruction buffer can reduce optimization opportunities when loads are not in the instruction buffer very long. Another difference is that our selection is done at compile-time. Although we need to introduce new instruction opcodes, we believe that the judgment made by the compiler is more accurate since it can analyze source-level program information and apply various heuristics.

Opcode Specifier	Meaning
ld_n	Normal loads
ld_p	Use address prediction
ld_e	Use early calculation

Table 1. Scheme specifier in the opcode.

3 Hardware Extensions

3.1 Instruction Set Modification

The compiler-directed approach to early address generation involves the introduction of at most three new load instruction specifiers. The names and functions of these specifiers are listed in Table 1. Opcode specifier `ld_n` is used for unpredictable load instructions to perform a normal load operation. The `ld_p` specifier directs the hardware to make an address prediction and allocate an address prediction table entry

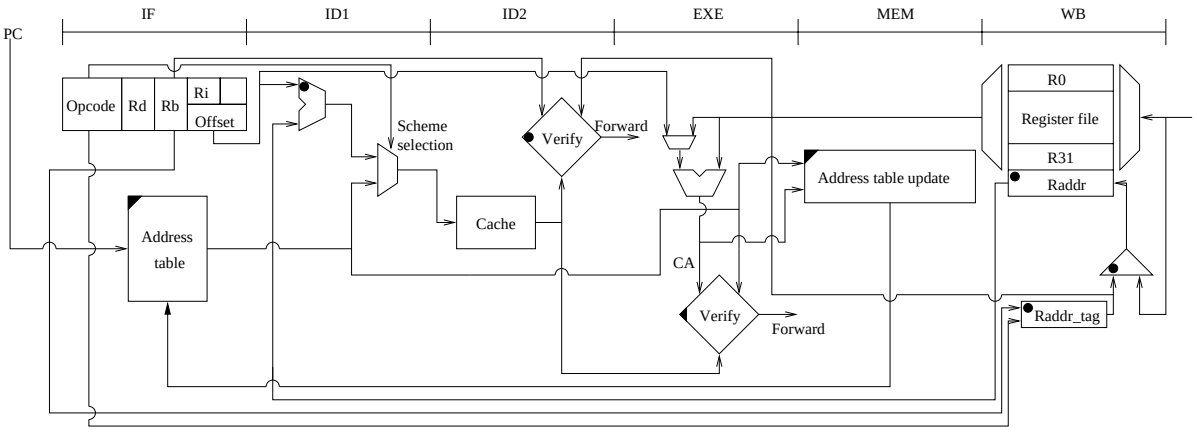


Figure 2. Experimental pipeline design.

for the load. The `ld_e` specifier selects the early calculation path. In addition, it stores the base register's content into a special addressing register R_{addr}. This is called the binding between R_{addr} and the general purpose registers. Further run-time details will be given later.

3.2 Pipeline Design

There are six stages in our experimental pipeline design with two decoding stages and one cache-access stage. Circuitry is added to the pipeline to support both early calculation and table-based prediction to generate the effective address speculatively. In the pipeline diagram depicted in Figure 2, units annotated with a dot are added for early calculation, while units annotated with a triangle are used for address prediction. When a load instruction is being decoded in the ID1 stage, its opcode can be used to select the appropriate scheme for early address generation.

3.2.1 Early Load-Address Calculation Path

One notable difference between our scheme and previous early calculation schemes [1][2] is that only one register content is buffered in a special one-entry cache which can be considered as a special addressing register called R_{addr}. Neither predecode information nor associative lookup are needed when retrieving R_{addr} since that is the only register being cached. Its content can always be used in computation when the instruction is being decoded and verified later when the decode information is available. Thus a dedicated full-adder instead of a carry-free adder [1] is affordable in our pipeline design. The binding between R_{addr} and one of the general purpose registers is setup by the `ld_e` instruction. For example, when instruction `ld_e Rd, Rb(N)` is seen, the content of R_b will be cached in R_{addr}.

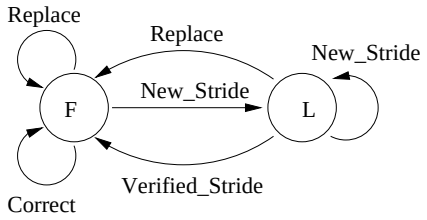
At the end of the ID1 stage, the information about the instruction type and the early generated address is available. If the instruction is decoded as `ld_e`, and if a data cache port is available, a speculative load will be dispatched to the cache based on the early calculated address. Though we use a full-adder to calculate the effective address, it is still possible to generate an incorrect result because there may be hazards on R_{addr}, or the binding has just been switched by the current load, or there are pending stores that write to the same location as the speculative load. By using the same notation in [1], the speculatively loaded data can be forwarded to subsequent dependent instructions if the following formula evaluates to true:

$\overline{R_{addr}\_Interlock} \wedge \overline{Mem\_Interlock} \wedge R_{addr}\_Hit \wedge Port\_Allocated \wedge DCache\_Hit$

As the formula shows, the fast-address calculation hazard in [1] is eliminated because a full-adder is used. And if data can be successfully forwarded, the latency for the load is reduced to 0. No recovery is needed since the formula is evaluated before the data is forwarded, and the mis-speculation penalty is an extra load issued to the memory system.

3.2.2 Table-Based Address Prediction Path

If a load's opcode is `ld_p`, its effective address is predicted based on its previous address plus an observed stride. Its previous address and observed stride are cached in a PC-indexed address table. Each entry in the address table has four fields: tag, predicted address (PA), stride (ST), and stride confidence(STC). If the table access is a miss, no prediction will be made for the load this time. Otherwise, the PA field will be speculatively used as the effective address to access the data cache in the ID2 stage, if a data cache port is available. Its verification will be performed at the end of the EXE stage by comparing the normally calculated



(a) State transition diagram

Arc	Condition	Operation		
		PA	ST	STC
Replace	tag1 != tag2	CA	0	1
Correct	PA == CA	CA+ST	N/C	N/C
New_Stride	PA != CA	N/C	CA-PA	0
Verified_Stride	CA-PA == ST	CA+ST	N/C	1

(b) Operations performed when state is in transition

Figure 3. State transition diagram for address table entries.

address (CA) and PA, together with checking memory interlocks. If the following formula evaluates to true, the load latency for this load is reduced to 1 cycle:

$\overline{Mem\_Interlock} \wedge Address\_Table\_Hit \wedge Port\_Allocated \wedge DCache\_Hit \wedge CA\_PA\_Match$

Again, no recovery is needed and the mis-speculation penalty is an extra load.

The corresponding table entry is updated in the MEM stage. An entry can be in one of the following two states: *functioning* or *learning*. The state transition diagram is shown in Figure 3a and the operations performed when the state is in transition are summarized in Figure 3b. A new entry is allocated if the PC-indexed probe is a miss. The new entry begins in the functioning state with the PA field set to CA, the ST field set to 0 and the STC bit set to 1. When the state is functioning and PA matches CA, the PA field in the entry is updated with PA+ST, where ST could be 0. When the state is functioning but PA differs from CA, ST is set to CA-PA and STC is set to 0, and the state becomes learning. When the state of the entry is learning, the value of CA-PA is compared with ST. If they match, the state becomes functioning, PA is updated with PA+ST, and STC is set to 1. Otherwise the state remains as learning and ST is set as CA-PA. In short, except for newly allocated entries, the stride confidence will not be built until the same stride is seen in two consecutive instances of loads.

4 Compiler Support

In our scheme, the selection between the dual early address generation mechanisms is guided by the compiler. Decisions are made based on the following two heuristics. Firstly, R_{addr} is an effective but scarce resource, so it should be reserved for those loads whose addresses are not linear. And secondly, the size of the address prediction table is desired to be small, so to reduce the chance of conflicts, non-linear loads should not be entered into the table.

The heuristics provided in this section are designed to classify loads into two categories: load-dependent loads and arithmetic-dependent loads. For load-dependent loads, if the addressing mode is register+offset, we would like to use the early calculation scheme, and for arithmetic-dependent loads, we prefer the table-based prediction scheme. For a load-dependent load whose base register is not used by many other loads, or whose addressing mode is not register+offset, we use `ld_n` as its opcode so that the R_{addr} register or the prediction table will not be contaminated.

We enhanced the IMPACT compiler Public Release 1.02 Beta 4 [9] to implement the compiler heuristics. These heuristics are applied after performing classical optimizations including function inlining, virtual register allocation, local/global constant propagation, local/global copy propagation, local/global redundant load elimination, loop invariant code removal, and induction variable elimination/strength reduction [10]. Our heuristics are dependent on these optimizations since they promote variables to registers. Without these optimizations, almost all loads will be termed as load-dependent loads thus the resultant classification will be useless. Though there are many other optimizations performed by the compiler, the above ones are the most related to the task of load classifications. We use different heuristics for cyclic and acyclic code, as described below.

4.1 Heuristics for Cyclic Code

When loops are analyzed, nested loops are sorted and inner loops are analyzed first. Loop analysis heuristics are designed based on the following observations:

1. After loop optimizations, loop invariant loads should have been moved out of the loop. The remaining loads in the loop will likely bring different values from iteration to iteration. If the loaded values are used for further dereferences, these dereferences are not likely to be linear so that the table-based prediction is not applicable.
2. Since arithmetic operations generate the results at the end of the EXE stage, in some situations it may impose the address-use hazards. Also, we assume that these arithmetic operations generate a series of linear values, which can be easily detected by our table-based prediction scheme. Therefore we will not se-

```
for (i=0; i<N; i++) {
      :
    .. = arr1[ind[i]]);
    .. = arr2[i]);
      :
}
```

(a)

```
_for:
   :
  op1  ld_p r4, r17(0)
  op2  lsl  r5, r4, 2
  op3  ld_n r6, r19(r5)
   :
  op4  ld_p r7, r18(0)
   :
  op5  add  r1, r1, 1
  op6  add  r18, r18, 4
  op7  add  r17, r17, 4
  op8  blt  r1, N, _for
```

(b)

```
while (p) {
      :
    .. = p->f1;
    .. = p->f2;
      :
    p  = p->next;
}
```

(c)

```
_while:
      :
  op11 ld_e r3, r2(0)
      :
  op12 ld_e r4, r2(4)
      :
  op13 ld_e r2, r2(8)
  op14 bne  r2, 0, _while
```

(d)

Figure 4. Code examples with new load instructions.

lect the early address calculation path for arithmetic-dependent loads to avoid the possible hazards and the contention for R_{addr}.

Given a loop, our heuristics proceed as follows:

1. For every load in the loop, add its destination register's specifier into set S_{load}.
2. For every arithmetic instruction found in the loop (*e.g.*, mov, add, sub), check the source registers' specifiers. If any of them appears in S_{load}, also add its destination register's specifier into S_{load}. Repeat this step until no more register specifiers are added to S_{load}. Now S_{load} contains the register specifiers whose contents are loaded from the memory or generated from a loaded value.
3. The appropriate early address generation scheme can be determined now. Load instructions whose base registers are in S_{load} are load-dependent loads. These loads are divided into several groups based on their base registers' specifiers. Loads in the largest group are assigned `ld_e` as the opcode, while loads in the remaining load-dependent groups are assigned `ld_n`. The remaining loads are arithmetic-dependent loads and `ld_p` is selected as the opcode.

We use the examples in Figure 4 to demonstrate how our heuristics work. After step 1, set S_{load} for the for-loop contains `r4`, `r6`, and `r7`. After step 2 the heuristics add `r5` to S_{load}. In step 3, `ld_n` is chosen for `op3`, since `r5` is in S_{load} but it uses the register+register addressing mode. The rest loads are arithmetic-dependent loads and thus assigned `ld_p`. For the while-loop, after step 1 S_{load} contains `r2`, `r3` and `r4`. The size of the R_2 group is three since it contains `op11`, `op12` and `op13`. Assuming no other groups contain more than three loads, R_{addr} will be reserved for R_2 thus `ld_e` is selected for `op11`, `op12` and `op13`.

4.2 Heuristics for Acyclic Code

Unlike cyclic code where previous instances of the same load can warm up the prediction table and R_{addr} for future executions, the heuristics for loads in the acyclic portion are designed under different principles. To use the table-based prediction scheme, only loads that load from absolute locations are attributed `ld_p`. The rest loads are grouped based on their base registers' specifiers. Loads in the largest group are assigned `ld_e`, and the remaining loads are assigned `ld_n`.

4.3 Address Profiling

Recent work in value prediction [11][12] use value profile information to improve the accuracy of hardware value prediction. Essentially instructions that are shown to be unpredictable are kept out of value prediction buffers, reducing the number of conflicts, resulting in more accurate prediction that requires smaller resources. In a similar way, address profile information can be used to support our compiler-directed early address generation.

In the compiler heuristics we assumed that remaining loads in the loop are likely to bring different values into registers in each iteration. However, this may be due to the conservative judgment made by the optimizer. Therefore some load-dependent loads are still accessing constant or strided locations. Assigning `ld_n` to these loads prevents them from utilizing the prediction table. The undesired result can be altered by performing address profiling. Our profiling model gathers statistics based on finding load instructions whose addresses are predictable by our stride-detection mechanism. Knowing that the profile information could adversely affect the program performance for different program inputs, it is used only to change a load classified as `ld_n` by our compiler heuristics to `ld_p` and nothing else will be overruled. The benefits of using profile information in early address generation are presented in Section 5.

5 Experimental Results

5.1 Methodology

The impact of compiler-directed early load-address generation on execution performance is evaluated in this section. The benchmarks used in this experiment consist of programs in SPEC92 and SPEC95 integer suites. MediaBench benchmark suite [13] was also evaluated for extending the performance results to embedded systems. All programs were compiled with traditional optimizations using the IMPACT compiler. All

Benchmark	Loads	% Static Loads			% Dynamic Loads			Prediction Rate	
	mil	NT	PD	EC	NT	PD	EC	NT	PD
008.espresso	83	17.25	50.08	32.67	18.10	74.52	7.38	92.65	77.92
022.li	9	19.76	30.10	50.14	21.59	35.37	43.04	54.56	95.19
023.eqntott	195	17.66	57.64	24.70	3.74	92.79	3.47	92.03	94.67
026.compress	13	9.12	85.04	5.84	26.01	73.74	0.25	83.07	95.11
072.sc	18	16.77	45.32	37.91	20.15	64.21	15.64	44.29	98.30
085.cc1	19	22.19	32.93	44.88	24.15	48.40	27.45	64.61	88.88
124.m88ksim	18	5.67	54.52	39.81	8.46	67.18	24.36	72.79	96.33
129.compress	6	9.29	82.51	8.20	26.83	70.49	2.68	75.40	97.72
130.li	33	19.16	29.79	51.05	13.96	35.98	50.06	78.94	88.96
132.ijpeg	237	22.05	28.88	49.07	32.50	63.37	4.13	33.16	91.98
134.perl	287	21.50	32.52	45.98	21.81	46.15	32.04	73.24	97.54
147.vortex	429	16.21	30.26	53.53	26.91	24.45	48.64	85.03	93.54
average	112	16.39	46.63	36.98	20.37	58.06	21.57	70.81	93.01

Table 2. Benchmark programs, load characteristics, and prediction characteristics.

results were generated using an emulation-driven simulator. The simulator performs a detailed timing simulation of in-order superscalar microprocessors and instruction and data cache memory systems.

The existing simulator was modified to investigate the early address generation methods presented in this paper. The simulated processor pipeline was changed to support address prediction in the first decode stage and load execution in the second decode stage of the pipeline. For address prediction, a parameterized address cache was added to generate the necessary load address predictions. Similar additions were made to the simulated pipeline to support early address calculation. Finally, a mechanism for supporting a limited number of cached base registers was also simulated.

The base architecture modeled in these experiments can fetch, decode, and issue up to 6 operations per cycle. The processor can execute these operations in-order up to the limits of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. The processor contains 64 integer registers and 64 floating point registers. The memory system consists of a 64K direct-mapped instruction cache and a 64K direct-mapped, non-blocking data cache, both with 64 byte block size. The data cache is write-through with no write allocate and has a miss penalty of 12 cycles. The branch prediction scheme is a 1K-entry BTB with 2 bit counters. The instruction set architecture and instruction latencies used match those of the HP PA-7100 [14] microprocessor (most integer operations have 1-cycle latency, and load operations have 2-cycle latency).

5.2 Speedup

In this section, we report performance results and reference characteristics of compiler-directed early address generation. In Table 2, the middle columns indicate the static and dynamic distribution of the load types classified according to the compiler heuristics of Section 4. The classifications are either *neither* (NT), *predict* (PD), or *early calculate* (EC). The final two columns of Table 2 indicate the prediction rates of those loads classified as NT and PD. These prediction rates were generated using a simulation methodology that performs individual operation prediction. Thus the predictions are not affected by the limitations of a prediction cache. An address prediction is made by emulating the state machine mentioned in Section 3 for capturing strided accesses for each load instruction.

The results of Table 2 indicate that the compiler heuristics of Section 4 are able to correctly classify loads as predictable. The average prediction rate of PD loads is 93.01%, while the NT loads have an average prediction rate of 70.81%. A few benchmarks have prediction rates for NT loads that are significantly above the average rate. The poor classification decisions reduced the potential performance speedup for *008.espresso* in the proposed approach, however, address profiling was able to correct the load classifications of the heuristics.

Figure 5 shows the performance speedup results for various system configurations. All performance speedup results are relative to a base architecture without any early address generation support. Figure 5a assumes a configuration where the hardware solely provides the table-based address prediction scheme with the capability of detecting strides. We varied the size of the table with 64, 128, 256 direct-mapped entries with and without compiler support. When compiler support is available, only loads identified as predictable by the compiler heuristics are allocated entries in the table. When the compiler support is not provided, all loads are assumed predictable. The outcome is as expected, larger tables provide higher speedups. When the size of the table is too small, both methods perform equally poorly. When a reasonably-sized table is given, the compiler-directed method outperforms the hardware-only method since fewer contentions for ta-

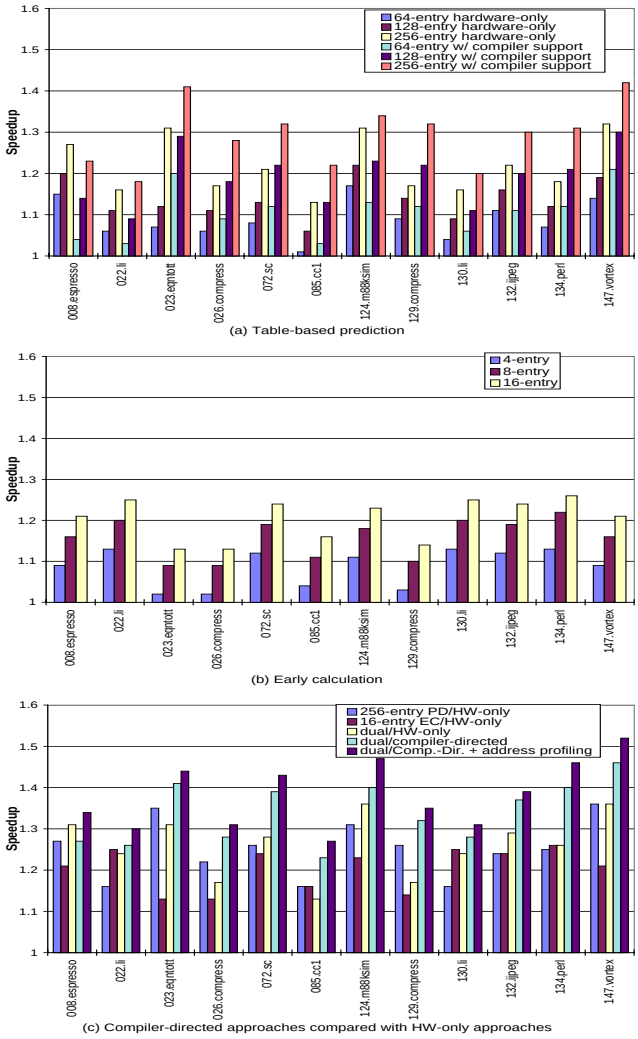


Figure 5. Performance speedup from early address generation.

ble entries are generated. Though not shown in the figure, the 1024-entry hardware-only approach was required to consistently surpass the performance of the 256-entry compiler-directed approach. In Figure 5b, we show the speedup when the hardware solely provides the early address calculation mechanism for early address generation. The number of cached registers are ranged from 4 through 16. Again as expected, better performance is obtained when more registers are cached. But the trend of speedup begins to slow down when the number of cached registers was doubled from 8 to 16 due to the address-use hazards. In Figure 5c, we show the performance numbers for the largest configurations of the hardware-only approaches in Figure 5a and 5b, together with the speedups generated by the proposed dual-path early address generation scheme. We generate three performance numbers for the dual-path scheme - without compiler support, with compiler heuristics, and with compiler heuristics plus address profiling. The early address generation hardware for these models consisted of a 256-entry prediction table and a single early address calculation register. When compiler support is not available, the selection of early address generation mechanisms is performed by the hardware at run-time. We use the selection heuristic proposed by Eickemeyer and Vassiliadis [3] which only allocates entries in the prediction table when there are register interlocks. When compiler heuristics are applied, different opcodes of load instructions perform the selection as explained earlier. Figure 5c also shows that address profiling can assist load classification.

From the charts in Figure 5, the speedup obtained through compiler-directed early load-address generation is significant. Comparing the two hardware-only mechanisms as shown in Figure 5a and 5b, it indicates that no method is overwhelmingly superior. When the majority of loads are arithmetic-dependent with constant or strided addresses, the prediction method wins, otherwise the early calculation method prevails. Figure 5c indicates that the dual-path approach provides better speedups over the hardware-only approaches, especially when address profiling is performed. Without address profiling, the compiler heuristics provided an average speedup of 34%. With address profiling the average speedup of our approach increased to 38%.

5.3 Prediction Rate

Our approach has an interesting advantage since it can adapt program analysis to refine its application mechanism. The results of Table 3 are generated using address profile information as a form of program analysis to direct the use of the early address prediction mechanism. The table is generated using a threshold value of 60%, which changes NT loads with greater than 60% address prediction rates into PD loads.

Table 3 indicates that address profiling can improve the classification of loads done by the compiler. Revisiting Figure 5c, the hardware-only dual-path early address generation scheme outperforms the scheme with compiler heuristics only for *008.espresso*. After performing profile-assisted load classification, 3.16% more of static loads or 15.71% more of dynamic loads are revised as PD loads which yield 6.88% more speedup, topping the hardware-only dual-path scheme. Comparing Table 2 and Table 3 indicates that many loads for these benchmarks were originally classified incorrectly as unpredictable. The new classifications of Table 3 demonstrates that the profile information can change the classification of these loads, thereby increasing the dynamic execution frequency of predictable loads by

Benchmark	Speedup	Static PD	Dynamic PD	Prediction Rate	
				NT	PD
008.espresso	1.34	53.24	90.22	49.20	82.06
022.li	1.30	31.12	39.19	16.37	95.66
023.eqntott	1.44	59.79	96.21	38.54	94.70
026.compress	1.31	85.77	83.12	41.43	95.08
072.sc	1.43	46.75	67.99	35.91	97.44
085.cc1	1.27	34.62	53.42	25.94	89.24
124.m88ksim	1.47	54.87	72.45	21.14	95.33
129.compress	1.35	83.06	74.74	27.89	97.86
130.li	1.31	31.15	38.95	23.05	89.87
132.ijpeg	1.39	31.80	64.52	29.18	91.72
134.perl	1.46	33.46	55.93	0.84	97.42
147.vortex	1.52	35.64	42.70	45.66	79.23
average	1.38	48.44	64.95	29.60	92.13

Table 3. Speedup, static and dynamic distribution of predictable loads, and prediction rates using profile information in load classification.

6.9%. This change caused an average 41.21% reduction in the prediction rate of NT loads, while slightly decreasing the average prediction rate of the predictable loads. The prediction characteristics for *134.perl* were most significantly affected by using profile information since the prediction rates for its loads were distinctly predictable or unpredictable. Further work needs to be done to understand ways of using these observations to refine the compiler heuristics of Section 4.

5.4 Embedded System Design

The compiler-directed early load-address generation method described in this paper may find the most suitable use in embedded systems. First of all, the compiler-directed approach requires changes in the instruction set architecture, and effectively excludes existing architectures. Since embedded architecture designs typically have shorter lifetimes and experience more architectural changes, such alterations to the instruction set may be more attractive. Secondly, embedded processors are typically in-order execution designs that have not been altered by the recent trend in dynamically scheduled microarchitectures for general purpose computing. The embedded design principle for in-order architectures is based primarily on memory system and hardware resource constraints. Thus, the compiler-directed scheme investigated in this paper is more likely to complement traditional embedded architectures than other load latency hiding techniques.

Finally, embedded systems face more restrictive space and power constraints than general purpose processors. Thus, even through progress in VLSI technology may eventually allow early address generation schemes implemented in hardware to achieve similar levels of performance to compiler schemes for general purpose machines, compiler-directed schemes may still be favored for embedded processor designs. Overall, compiler based approaches advocate limited and specialized hardware resources, a common theme in embedded system design.

In order to evaluate the compiler-directed early address generation method for embedded systems, the MediaBench benchmark suite was studied. These programs characterize applications that typically execute on embedded systems. Table 4 shows the load characteristics for the early address generation scheme for these programs. In general the MediaBench applications exhibit more loads with address predictability than the loads in the SPEC benchmarks of Table 2. This is indicated by a 21% difference in the percentage of dynamic load executions that were classified as predict. Overall, such loads of the MediaBench applications also have a slightly higher prediction rate. Since loads in MediaBench make up a smaller portion of executed instructions, the average speedup is 19%.

6 Conclusions

In this paper we present a novel early address generation scheme which accommodates both early address calculation and table-based address prediction with stride detection. Through analysis of load instructions in various program contexts and through experimentation, we have shown that neither approach alone captures all the important cases across a variety of loads. The key to good performance is to use the proper type of early address generation for each load. The compiler is in the best position to make such determinations based on its ability to analyze code and apply heuristics for using the mechanism. The determination can be refined statically by using address profile information. Nevertheless, even with simple heuristics, the compiler-supported schemes performed better than the hardware-only approach. Furthermore, these results were achieved with significantly reduced hardware overhead. These results are particularly interesting for embedded processors since hardware space and power constraints dominate design decisions.

Our results indicate that with compiler support, an average speedup of 34% is achieved with one special addressing register and a 256-entry address prediction table. Hardware-only approach generates only 26% speedup for the same dual-path model. Our speedup number is improved by an average of 4% after enhancing load classifications with address profile information. Overall, these results demonstrate that our proposed model is effective in reducing the latency of load instructions and improving performance.

Further enhancements to our heuristics can be made by performing aggressive compiler analysis. Though

Benchmark	Loads mil	% Static Loads			% Dynamic Loads			Prediction Rate		Speedup
		NT	PD	EC	NT	PD	EC	NT	PD	
G.721 Decode	28	16.67	36.90	46.43	18.16	66.73	15.11	39.67	81.47	1.15
G.721 Encode	27	16.87	37.35	45.78	18.46	66.41	15.13	39.07	78.21	1.15
EPIC Decode	1	11.88	62.62	25.50	9.73	78.34	11.93	55.14	99.02	1.22
EPIC Encode	6	7.20	40.06	52.74	3.43	96.46	0.11	39.86	86.20	1.23
Ghostscript	1	11.41	29.43	59.16	17.79	48.06	34.15	52.34	84.18	1.11
GSM Decode	5	3.07	35.58	61.35	0.44	98.34	1.22	31.64	76.48	1.21
GSM Encode	20	4.19	34.16	61.65	1.05	96.55	2.40	38.20	94.04	1.25
MPEG Decode	21	8.21	73.31	18.48	3.48	94.48	2.04	27.19	73.31	1.19
PGP Decode	1	9.95	69.94	20.11	0.29	98.91	0.80	29.73	98.58	1.27
PGP Encode	1	9.95	69.94	20.11	6.73	77.28	15.99	26.56	71.08	1.15
RASTA	2	19.30	44.38	36.32	12.39	82.89	4.72	36.69	91.32	1.21
ADPCM Decode	1	21.43	50.00	28.57	39.99	59.93	0.08	16.21	81.03	1.16
ADPCM Encode	1	28.57	42.86	28.57	33.33	66.60	0.07	16.21	86.59	1.14
average	8	12.98	48.19	38.83	12.71	79.31	7.98	34.50	84.73	1.19

Table 4. MediaBench programs, load characteristics, prediction characteristics, and speedup.

we applied function inlining to remove frequently executed function calls in the loop, any remaining function call may prevent an important load from being migrated out of a loop. This will cause some arithmetic-dependent loads to be identified as load-dependent loads and prevent them from being predicted. Thus, the results of more aggressive analysis will provide many opportunities to further improve overall performance by better utilizing the proposed compiler-directed early address generation scheme.

7 Acknowledgments

The authors would like to thank John Gyllenhaal, Matt Merten and all the members of the IMPACT compiler team for their valuable comments. This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948, Intel Corporation, Advanced Micro Devices and Hewlett-Packard.

References

[1] T. M. Austin and G. S. Sohi, "Zero-cycle loads: Microarchitecture support for reducing load latency," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 82–92, December 1995.

[2] C. Chen and A. Wu, "Microarchitecture support for improving the performance of load target prediction," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 228–234, December 1997.

[3] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 27, pp. 547–564, July 1993.

[4] M. Golden and T. N. Mudge, "Hardware support for hiding cache latency," tech. rep., University of Michigan, February 1993.

[5] J. Gonzalez and A. Gonzalez, "Speculative execution via address prediction and data prefetching," in *Proceedings of the 1997 International Conference on Supercomputing*, pp. 196–203, July 1997.

[6] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[7] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 218–227, December 1997.

[8] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The performance potential of data dependence speculation & collapsing," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 238–24, November 1996.

[9] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[10] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[11] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 259–269, December 1997.

[12] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 270–280, December 1997.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[14] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.