SMLNJ: Intel x86 back end
Compiler Controlled Memory

Lal George
Rm 2c478, Bell Labs
700 Mountain Ave
Murray Hill, NJ 07974

October 4, 1999

This note describes the code generation algorithm used for the Intel x86, introduced in version 110.16. The standard Chaitin graph coloring register allocation cannot be used directly for machines with few registers, as all temporaries wind up being spilled, making for a poor allocation[Cha82]. Thus, for the x86, the conceptual model of the architecture has been extended with a set of memory locations that are treated as registers. The net effect is one where temporaries are computed into memory locations and passed as arguments to functions. The use of these memory locations managed in this way can be as fast as using registers, where the register allocation algorithm is indirectly taking the hardware register renaming mechanism into account.

1 Intel micro-architecture

The intel architecture has gradually embraced many of the ideas popular in RISC designs such as a pipelined organization, multiple instruction issue, register renaming, and out-of-order execution. Yet the interface to the architecture, presented by the set of registers and instruction set, has remained virtually unchanged. As always, an understanding of how registers are used and instructions executed on the micro-architecture are an essential prerequisite to generating good code. What makes the intel architecture particularly challenging is the small number of registers.

This paper focuses on the Pentium II, an 11-stage pipeline (Figure 1) with out-of-order execution and register renaming.

IF1..3	fetch, alignment, and branch prediction.
DEC1..2	instruction decoding
RAT	regster alias table and allocator
ROB	move μ-ops into reorder buffer
DIS	dispatch out-of-order
EX	execute
RET1..2	retire

Figure 1: Intel Pentium II pipeline

The first five stages fetch and decode x86 instructions translating them into equivalent RISC operations called μ-ops. A register allocation table (**RAT**) is then consulted to remap the eight external registers into one of 40 internal registers. The 128 bit μ-ops are then dumped into a 40 entry circular buffer (**ROB**) where they wait their turn to be executed. The instructions at this point can be executed out-of-order and speculatively based on data dependencies, branch predicition, and the availability of functional units. When ready to execute, the instructions are queued at the functional units in the dispatch stage (**DIS**) and finally execute at the functional units. The μ-ops are retired in strict program order in the last two stages (**RET**) implying that the entry in the **ROB** is not reclaimed until all prior instructions have been retired.

X86		μ-ops	
%eax	= MEM[y]	load	MEM[y],r25
%eax	+= MEM[z]	load	MEM[z],r26
		add	r25,r26,r25
MEM[x]	= %eax	lea	MEM[x],r27
		store	r25,(r27)
%eax	= MEM[u]	load	MEM[u],r28

Figure 2:

2 Register renaming

It is useful to understand how register renaming works on the Pentium II. Due to the limited number of registers, the style of code generation is one where a register is loaded from memory, used in some computation with a second operand in memory, and the register result written back to memory. For example, to compile `x := y + z; ...` the code that gets generated is shown on the left-hand-side of Figure 2.

Because of the dependency on `%eax`, there is no instruction level parallelism exhibited by this code sequence. However, when coverted to μ-ops the first use of `%eax` is rewritten to use the internal register `r25` via register renaming. The second x86 instruction is translated to two μ-ops, one to load the second operand into an internal register `r26`, and another to perform the register-register addition. All of a sudden there is a large amount of instruction level parallelism. The `lea` and the first two `load` μ-ops can each be issued in parallel. Once the data has arrived the `add` and `store` μ-ops can be issued in parallel, due to the internal feed-forwarding mechanisms.

3 Pentium II optimizations

The intel optimization manual suggests several ways to improve the performance on the architecture by (for example): paying attention to the branch prediction algorithm, aligning data, avoiding complex instructions, reducing the number of branches, etc. However, many (if not all) of these are already performed by default in most compiler back ends.

I plan to investigate four optimizations specific to the Pentium II:

Instruction scheduling: By rearranging the order of instructions generated, the processor may find many more independent instructions in its reorder buffer. Given the small number of registers, and the aggressive out-of-order execution engine, I am doubtful about how effective this might be for most programs.

Instruction decoding: The speed at which the reorder buffer gets filled is dependent on how fast the processor can decode instructions. The two **DEC** stages in the pipeline can decode three x86 instructions in parallel — only one of which is allowed to be complex. Spreading apart complex instructions would therefore fully utilize the decoding logic. Earlier, preliminary experiments along these lines by Allen Leung, on the SML/NJ compiler found this to ineffective.

Compiler controlled memory: There is a mechanism in the processor where frequently accessed memory locations can be as fast as registers. In Figure 2, if the address of `u` is the same as `x`, then the last `load` μ-op is a `nop`. The internal value in register `r25` is forwarded to register `r28`, by a process called write-buffer feedforwarding. That is to say, provided the `store` is pending or the value to be stored is in the write-buffer, then loading form a memory location is as fast as accessing external registers.

Transparency control: If the `store` μ-op has been retired, then all bets are off with write buffer feedforwarding. A subsequent `load` μ-op must go through the load pipeline and pay at least a 3 cycle penalty. However, if a value is loaded into an external register such as %eax, and kept there, then the register renaming mechanism will allocate a new internal register which will be reserved for %eax until the external register is redefined. This suggests that keeping values in external registers for as long as possible may be a valuable optimization.

This paper does not deal with *transparency control*, but focuses on *compiler-control memory*. Besides being able to treat memory locations as registers, the technique I have developed is essential for graph coloring to work in the first place. The technique is similar in spirit to "Compiler-control memory" described by Cooper and

Harvey[CH98], however substantially different in implementation.[1]

4 Register allocation

An important function of a register allocator is to target registers so that the result of an instruction is in a destination register where it might be required later[GA96] – as an argument to a function call, for example. One can go a step further and use the register targeting mechanism in the allocator to determine a good set of registers for parameter passing, thereby specializing individual calling conventions. This optimization is particularly important for call intensive languages.

Because there are very few registers on the x86, the recommended (or standard) calling convention is one where everything is passed in memory using the stack, illustrated in the program below and the generated code shown to the right.

```
f(x,y,z)                      f:
  ...                                 ...
  call g(0,x,y);                  push(y)
                                  push(x)
                                  push(#0)
                                    call(g)
  ...                                 ...
  call h(z,y,x,1);                push(1)
                                  push(x)
                                  push(y)
                                  push(z)
                                    call(h)
```

A register allocator would try and assign the calling convention for `g` and `h` so that none of the `push` instructions are required.

5 Register allocation for the x86

Directly applying Chaitin style graph coloring register allocation to the x86 results in a disaster as everything spills.

[1] I came up with the idea independently and probably around the same time that Cooper and Harvey did, as I had a working implementation of the idea when their paper was published.

Further, the heuristic used to target registers does not perform well because of the small number of available registers compared to the average number of simultaneously live variables.

Our strategy is to pre-allocate a small set of memory locations that will be treated as registers and managed by the register allocator. With this strategy, it then becomes conceivable that the calling conventions for `g`, `h`, and `f` above, pass their arguments in one of these memory registers, and no extra shuffling or `push`s are required.

In summary, the register allocator is now extended to allocating the *right* memory location (or memory register) to temporaries in the program.

6 Register allocation framework

All my work is done in the context of MLRISC[Geo] a customizable and retargetable code generation framework. In MLRISC, the register allocator can be depicted as the black box shown in Figure 3. It takes as input a

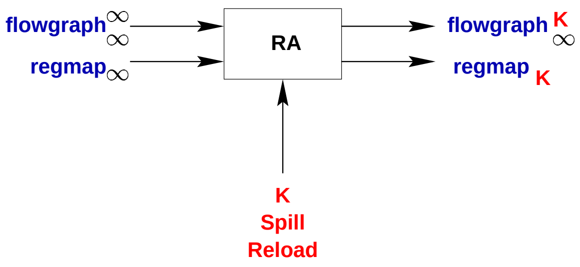

Figure 3: Register allocation framework

flowgraph $\texttt{flowgraph}^{\infty}_{\infty}$ where the subscript says that the flowgraph mentions an infinite number of registers and the superscript says that the flowgraph will run on a machine with an infinite number of registers. The instructions in the flowgraph are never rewritten, but the result of register allocation is reflected in the table `regmap`. The subscript in $\texttt{regmap}_{\infty}$, says that temporaries in the program are mapped onto an infinite number of registers. Therefore before register allocation `regmap` is the identity function.

After register allocation, the result is $\texttt{flowgraph}^{K}_{\infty}$ and $\texttt{regmap}_{K}$ indicating that the flowgraph mentions an infinite number of registers but can run on a machine with K registers, after the resulting $\mathtt{regmap}_K$ has been applied to all the instructions in the flowgraph.

The register allocator RA, is specialized with the number of registers (or colors) K, and callback functions `spill` and `reload` that indicate how instructions are to be spilled and reloaded. In general these callbacks will take an instruction, a register to spill or reload, and will return a list of replacement instructions.

7 RA32

The x86 has eight external registers of which one (esp) is dedicated and cannot be used for other purposes. I first pretend that there are 32 registers, where the extra 24 will be implemented in memory, and do a usual round of register allocation as shown in Figure 3, but where K=32. I call this register allocator RA32. The result from RA32 is $\mathtt{flowgraph}^{32}_{\infty}$ and $\mathtt{regmap}_{32}$, indicating that the original flowgraph has been sufficiently simplified (via spilling) so that it can run on a machine with 32 registers. In this mode, the register targeting (via coalescing) is usually quite successful because of the large number of registers.

Consider a program:

```
t290 := t290 + t170
...
t300 := t300 * t290
```

When compiled to the x86, the original instructions of $\mathtt{flowgraph}^{\infty}_{\infty}$ and the result of applying $\mathtt{regmap}_{32}$ after register allocation to obtain instructions in $\mathtt{flowgraph}^{32}_{\infty}$ is shown below:

$$\left.\begin{array}{ll} addl & t_{170}, t_{290} \\ \dots & \\ mull & t_{290}, t_{300} \end{array}\right\} flowgraph^{\infty}_{\infty}$$

$$\Big\downarrow\ \mathtt{regmap}_{32}$$

$$\left.\begin{array}{ll} addl & r_{17}, r_{29} \\ \dots & \\ mull & r_{29}, r_{30} \end{array}\right\} flowgraph^{32}_{\infty}$$

The temporary $\mathtt{t}_{170}$ has been mapped onto $\mathtt{r}_{17}$, $\mathtt{t}_{290}$ onto $\mathtt{r}_{29}$, and so on. The result of applying $\mathtt{regmap}_{32}$ is nonsensical because $\mathtt{r}_{17}$ is really a memory location, and we may have created illegal instructions. The x86 does not allow two memory operands for example.

We could consider doing another round of register allocation on this to fix it. The inputs to the second register allocator are $\mathtt{flowgraph}^{32}_{\infty}$ where illegal uses of memory registers have been removed, and $\mathtt{regmap}_{32}$. The new value of K is eight. What might be generated for the above is something like:

$$\begin{array}{lll} addl\ \ r_{17}, r_{29} & \Rightarrow & \begin{pmatrix} movl & (r_{17}), \%eax \\ addl & \%eax, (r_{29}) \end{pmatrix} \\ & \dots & \\ mull\ \ r_{29}, r_{30} & \Rightarrow & \begin{pmatrix} movl & (r_{29}), \%ebx \\ mull & \%ebx, (r_{30}) \end{pmatrix} \end{array}$$

where $(\mathtt{r}_{17})$ denotes the effective address for the memory register $\mathtt{r}_{17}$, etc. If we recall how the register renaming works, the result of the first movl will be renamed into one of the forty internal registers. Let's suppose it gets mapped onto internal register 17, which is certainly possible since we do not have more than 32 registers simultaneously live at any one time. The addl instructions then gets translated into a load μ-op (to load the contents of the memory register $\mathtt{r}_{29}$; an add μ-op to do the addition, and finally a lea and store μ-ops to write back the result. Again we can assume that the effective address for $\mathtt{r}_{29}$ is put into internal register 29, which is certainly possible for the same reasons. The effect is that the external memory locations are getting mapped directly onto the internal registers, and we are in some sense controlling the pressure on the internal registers.

If the instruction

```
movl    (r29), %ebx
```

occurs soon after the addl that defined the memory location being used, then the write buffer feedforwarding takes place and the same internal register 29 can be used. In effect, the memory locations are acting as fast as registers, and we have targetted our allocator to the memory locations as frequently as possible.

8 Problem with RA32

There is a problem with RA32. Consider the program:

```
x := A[i]
...
y := A[i+1]
...
z := A[i-1]
...
A[i] := ...
```

Let's assume that the result of RA32, maps A onto r_{27}, i onto r_{11}, and x onto r_{13}. The instruction that would then be generated for `x := A[i]` would be:

```
movl   0(r27,r11,4), r13
```

which uses the indexed addressing mode. The address of the memory location is r_{27}+r_{11}*4. This is illegal as all the components of an indexed effective address (except the scaling factor) must be one of the eight physical x86 registers. Thus we would have to rewrite this into something like:

```
movl   (r27), t1
movl   (r11), t2
movl   0(t1,t2,4),t3
movl   t3, (r13)
```

before doing the next round of register allocation. And similarly for the other array operations in the program.

Every time the array A and index i are used in an index addressing mode, they must be moved into temporaries, and another round of register allocation performed. This is unfortunate. Perhaps, the base array A is used so often that one would have liked to use a real register instead of a memory location — for the purpose of code size at the very least. RA32 does not know the difference between the first eight registers and the remaining 24 implemented in memory. It assumed that all the 32 registers were the same.

9 RA32+RA8

The problem with the above scheme is that most of the damage has already been done in RA32 before the second register allocation takes place. Instead we want something like that shown in Figure 4.

RA32 is used to simplify the original flowgraph so that it will run on a machine with 32 registers. In other words $\texttt{flowgraph}^{32}_{\infty}$ is a modification of the original flowgraph by the insertion of spill code. In particular, the cheapest temporaries to spill would have been removed. Again, copy-propagation and coalescing are very successful.

We then pass this ($\texttt{flowgraph}^{32}_{\infty}$) together with the identity regmap, which says that all the remaining temporaries (an infinite number of them) map onto themselves; and then we perform register allocation with K=8 (RA8). In RA8 lots of temporaries are going to spill, and copy-propagation is going to fail miserably because of the shortage of registers. But $\texttt{regmap}_{32}$ is used to support the spill and reload decisions in RA8. That is to say if t_{300} spills but was assigned to r_{21}, then we spill t_{300} into the memory location for r_{21}. We are guaranteed to find it free from the calculations done in RA32. Further, if t_{300} and t_{400} were each assigned to register r_{17} and were coalesced in RA32, then they would both spill into the memory location for r_{17}. Thus we have copy-propagation taking place in memory, and all the nice decisions that RA32 made being carried over to RA8.

More importantly, if RA8 determines that the base address of an array (for example) should be kept in a register (because it is costly to spill), then the information is $\texttt{regmap}_{32}$ is not used for this temporary but one of the eight external registers gets assigned to that temporary.

It could happen that a temporary assigned to one of the physical registers in RA32, must be spilled when going to RA8. Thus there must be a reserved spill area for the physical registers as well.

10 Floating point

Floating point is implemented in a similar way. I assume 32 memory locations that act as floating point registers, and use the internal floating point registers as a stack — the way they were designed to be used. Sethi-Ullman numbering[ASU85] is used to manage the stack, and graph coloring is used to manage the memory locations. I have not implemented the use of the fxchg instruction as recommended by the Intel manual.

11 Results

This scheme is definitely better than applying RA8 alone as everything would spill in the latter. We compare the

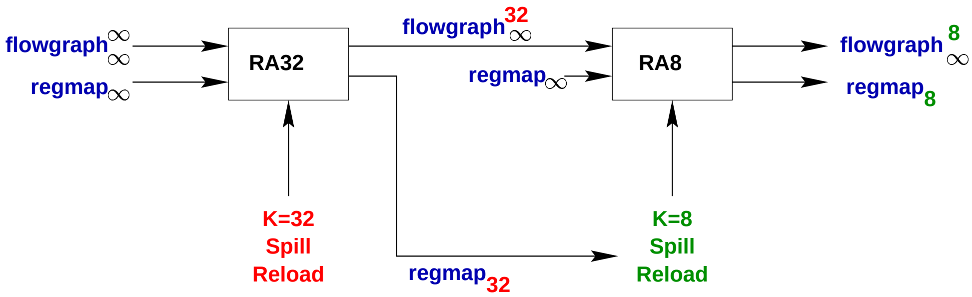

Figure 4: Ra32 + ra8

result of the new code generator will a hand-crafted one used in the SML/NJ compiler in version 110.0.3. the results are shown in Table 1.

Name	110.0.3	RA32+RA8	Speedup
tsp	7.55	6.49	16.39%
logic	5.45	4.87	11.84%
fft	1.11	0.81	37.89%
barnesHut	3.89	3.04	27.64%
ray	3.93	3.26	20.28%
mandelbrot	1.32	0.56	134.22%
simple	2.85	2.76	3.02%
vliw	2.20	1.78	23.64%
mlyacc	0.49	0.47	3.86%
lexgen	0.97	0.91	6.99%
knuthBendix	1.05	0.78	34.98%
life	0.14	0.12	13.51%
boyer	0.25	0.22	13.74%

Table 1: Performance results

I would like to predict that programs with tight loops would show the best improvement with this scheme, but have not verified it. The predicition is certainly validated in the mandelbrot benchmarks which improves by over a factor of two.

Compilation time in `RA32+RA8` is uniformly increased by 40%.

12 Related Work

Cooper and Harvey describe a scheme that should generate similar results to those presented in this paper. They use a modified version of RA8 where after the first round of allocation, anything that spills is allocated into the compiler controlled memory (CCM). The original interference graph is consulted to determine the assignment of spill nodes into CCM. They don't mention coalescing explicitly, but it would be trivial to eliminate moves between two nodes that have been spilled and allocated in CCM. Nodes that cannot be spilled into CCM are written out to main memory or the current activation frame. In other words graph coloring register allocation is performed using the graph made up of spill nodes. Subsequent rounds of register allocation include nodes and edges corresponding to slots in CCM in the interference graph, as well as ordinary nodes. The CCM related nodes and edges are ignored until all the new spill nodes have been accumilated in this round. The combination of spill nodes and CCM slots and edges are then used to determine the location of the new spill nodes in CCM. Again colaescing could be used to eliminated moves between newly spilled nodes and nodes that have previously been assigned slots in the CCM. While the final results ought to be similar to ours, we believe our algorithm is more elegant as it can be easily implemented by plumbing together an existing graph coloring register allocator.

13 Conclusion

I have shown an effective way to apply graph coloring register allocation to a machine with few registers. In particular, on the Intel Pentium II, the technique can be used to indirectly allocate the internal renamed registers.

References

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[CH98] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proc. of the 8th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 1–11. ACM SIGPLAN, November 1998. Sigplan notices volume 33, number 11.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Proceeding of the ACM SIGPLAN '82 Symposium on Compiler Construction.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM transacations on programming languages and systems.*, 18(3):300–324, May 1996.

[Geo] Lal George. Mlrisc home page. http://cm.bell-labs.com/cm/cs/what/smlnj/MLRISC.