Unified Management of Registers and Cache Using Liveness and Cache Bypass[1]

Chi-Hung Chi

Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

Hank Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

In current computer memory system hierarchy, registers and cache are both used to bridge the reference delay gap between the fast processor(s) and the slow main memory. While registers are managed by the compiler using program flow analysis, cache is mainly controlled by hardware without any program understanding. Due to the lack of coordination in managing these two memory structures, significant loss of system performance results because:

- Cache space is wasted to hold inaccessible copies of values in registers.
- Inaccessible copies of values replace those accessible ones from cache.
- Despite the fact that register allocation has long recognized the benefits of live range analysis, current cache management has completely ignored live range information.

In this paper, we propose an unified management of registers and cache using liveness and cache bypass. By using a single model to manage these two memory structures, most redundant copies of values in cache can be eliminated. Consequently, bus traffic and memory traffic in data cache are greatly reduced and cache effectiveness is improved.

Keywords: cache, register, live range, cache bypass, unified management.

1. Introduction

In current computer memory system hierarchy, registers and cache are both used to bridge the reference delay gap between the fast processor(s) and the slow main memory. While registers are managed by the compiler using program flow analysis, cache is mainly controlled by hardware without any program understanding. Due to the lack of coordination in managing these two memory structures, significant loss of system performance results because:

- Cache space is wasted to hold inaccessible copies of values in registers.
- Inaccessible copies of values replace those accessible ones from cache.
- Despite the fact that register allocation has long recognized the benefits of live range analysis, current cache management has completely ignored live range information.

This causes busy redundant memory traffic in cache and decreases system performance substantially. In load/store VLSI processor designs such as RISC architecture [Pat85] [HeJ83] [Kat83], this

1. This work was completed and the first version of this report was written up before Chi-Hung Chi joined Philips Laboratories.

problem becomes more serious because of the limited on-chip cache size and the high off-chip to on-chip memory access ratio [Hil83] [AgC87] [KaM87].

In this paper, we present an unified scheme for managing registers and cache taking full advantage of live range analysis. Redundant memory traffic in data cache due to inaccessible copies of values are eliminated and cache performance is improved. Throughout the whole discussion of the unified management scheme of registers and cache, a data cache with line size of one is assumed. This assumption is justified by the fact that small line size (e.g. one) is always preferred for data cache [ChD89] [Lee887].

The outline of this paper is as follows. Section 2 reviews the general characteristics of registers and cache. Differences between these two memory structures from compiler's view are also summarized. In Section 3, basic concepts toward unifying management of registers and cache are discussed. The proposed unified management scheme is then proposed in Section 4. Implementation of this scheme (both hardware and software) is also discussed in this section. In Section 5, simulation results of memory traffic reduction in data cache is presented. Finally, this paper concludes in Section 6.

2. Registers versus Cache

In order to devise a coordinated scheme for management of registers and cache, it is first necessary to develop a better understanding of the differences and similarities between these two types of buffer memory.

2.1. Registers

2.1.1. Concepts of Registers

Registers, or a "register file", constitute a relatively small, fast, local memory residing in an address space distinguished from that of main memory. The structure of a register memory cell is given in Figure 1.

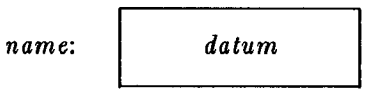

Figure 1. Structure of Register Memory Cell

Since registers are the absolute top of the memory hierarchy (typically with cache just below), register access time is the fastest of all memory systems in a computer and there are typically fewer memory cells in a register file than there are cells in any other level of the memory hierarchy. Each register is usually one word wide, with a total of perhaps 16 or 32 words in the register file.

By placing a value in a register, one can reap at least four benefits:

[1] The fast access time of values in registers reduces latency.

[2] A reference to a register typically does not interfere with references along the path(s) to main memory, thereby effectively increasing usable bandwidth to main memory.

[3] Typically, the predictability of register references aids in compile-time optimization of code and simplifies hardware. Optimizations are aided in that reference times can be known at compile time; hardware is simplified in that register references in most machines cannot cause pipeline bubbles.

[4] Because register names are typically shorter than memory addresses, referencing values in registers actually decreases the required instruction-fetch bandwidth — even though registers typically cannot hold instructions.

2.1.2. Register Allocation

Register allocation — the mapping of values in a program to physical registers — is traditionally handled by the compiler. Most "good" register allocation schemes are based on one of two principles:

- Usage count: the reference *frequency* of each value is used as the main criteria for allocating a register for a value. Values with higher reference frequencies should have higher priority to be in registers [Fre74].
- Graph coloring and spilling: a live-range interference graph is constructed using data flow or dependence analysis. In this graph, each node represents a value[2] and each arc

2. Instead of using a node for each value, each node may represent a variable. This is easier to implement, but degrades performance by introducing false dependencies where a variable is

linking two nodes represents the fact that the two values have overlapping lifetimes, i.e., are simultaneously live. A mapping of values to registers is found by coloring this graph with n colors, where n is the number of registers available (hence, each color represents a particular register). Various heuristics have been proposed for finding an n-coloring [ChA81] [Cha82] [Cho83] [ChH84]; should the algorithm fail to find an n-coloring, some values will be placed in memory (spilled from registers) to simplify the graph so that an n-coloring can be found.

Both these basic approaches share a number of characteristics:

- Rather than using the program's sequencing of references, a partial order derived from that sequence is used for allocation.
- Register allocation is visible to the compiler and, in some cases, also to the programmer.
- Binding of value to a name is defined at compile-time.
- It is understood that defining live range in terms of values rather than in terms of variables is beneficial.
- Conventional registers can only hold data, not instructions.

2.1.3. Limitations of Registers

The most important limitation of registers, is that, for most programs, many values cannot benefit from being kept in registers. Although it is true that sometimes a value cannot be kept in a register because the hardware provided too few registers, even given an infinite number of registers, a large fraction of the values computed within any program should not be kept in registers. To understand why some values should not be kept in registers, one must understand a little bit of compiler flow analysis.[3]

used to hold several different values.

3. The description given here of the ambiguous alias problem is a gross oversimplification intended only to give an intuitive introduction to the problem. This issue is currently one of the richest research areas within compiler technology; more detailed discussions of this problem appear in [All83], [Bur84], [BuC86], [All86], [Ste86], and [Die87].

Suppose a particular segment of a program refers to two names, one called α and the other called β. If one of α and β is a pointer, or one is a call-by-address argument to this routine and the other is a variable which was accessible in the caller's scope, or both are elements of the same array (such as *a[i]* and *a[j]*), etc., then it is possible that even though α and β look like different names, they refer to the same object. In other words, changing the value of one might change the value of the other, i.e., α and β might be aliases for the same object.

If compile-time analysis can prove that α and β cannot be aliases for the same object, then α and β can each be assigned to a register and each can be kept there indefinitely. Instead, if the compiler can prove that α and β are always aliases for the same object, then α and β are assigned to share a single register, and again the object can be kept in a register indefinitely. However, if the compiler isn't sure if α and β refer to the same object, or if α and β only sometimes refer to the same object, we say that α and β are ambiguously aliased to each other.

At this point, it is useful to point out that compile-time analysis techniques for determining if α and β are aliases for each other are, at best, complex to implement and easy to confuse. Confusion results in the "safe" assumption that α and β are ambiguously aliased to each other. In addition, in many cases it is ***theoretically impossible*** for the compiler to determine whether α and β are aliased, in which case the compiler must again assume that they are ambiguously aliased. A good example of such a case is determining whether *a[i]* and *a[j]* are aliased in code like Figure 2.

```
read (i, j);
a[i+j] = a[i] + a[j];
```

Figure 2: Example of Compile-Time Unsolvable Aliasing Problem

If the compiler's best "guess" is that α and β are ambiguously aliased, then placing either value in a register will require "flushing" that register whenever either α or β is stored into. This "flushing" is usually needed so often that the cost of referencing α and β from registers is actually greater than the cost of referencing them from main memory, hence, placing α and β in registers would degrade, rather than improve, performance.

2.2. Cache

2.2.1. Concepts of Cache

A cache is a small memory holding rapidly accessible copies of values from main memory. Rather than having a distinct namespace as registers do, cache contents are addressed associatively by main memory addresses. The conceptual structure of a cache memory cell is shown in Figure 3.

Figure 3: Cache Memory Cell Structure

Residing just below registers in the memory hierarchy, both access time and number of memory cells for a cache are between those of registers and those of main memory. Each cache cell (i.e., cache line) usually holds between one and 64 words, not counting the address tag. A typical cache implemented on the processor chip contains 128 to 256 words; a cache implemented using separate logic can be as large as 64K words, but is more often between 2k and 8K words.

Of the four benefits listed for placing a value in a register, however, in general, caches only insure [1]: a reduction in access latency. Benefit [2], which is based on the lack of interference between register and main memory data paths, does not hold for a traditional cache, except in that other processors in a multiprocessor system typically would not see interference from cache references on a particular processor's cache.

The predictable reference time for a register reference, benefit [3], is not echoed in cache reference because of the concept of a cache miss. Some would argue that, given a large enough cache, the probability of having a cache miss can be made arbitrarily low; however, we believe this misses the point. One reason we disagree with the very-large cache argument is that the access speed of a memory is related to the size of its address space (e.g., if one can fit the cache on the processor chip, it will probably function much faster than if it is referenced across several chips). Another reason is that the cost of implementing an arbitrarily large cache is also arbitrarily large — it isn't very cost effective. In any case, unless the cache is as large as the entire virtual address space of the machine, one will occasionally suffer a cache miss, and this implies that extra hardware/software effort must be made to cope with this situation.

Benefit [4] is based on the reduction of required instruction-fetch bandwidth due to use of short names in referencing values. This cannot be applied to cache because the register correspondence between short names (register numbers) and long names (memory addresses) must be explicitly established by register *Load* and *Store* instructions, whereas the mapping in a cache is unknown to the software. In other words, a compiler cannot tell which cache line of a conventional cache will hold a copy of a particular value it is referencing — hence, it cannot use the cache line number to address the value. The desired value might not even be in the cache, either because it has not yet been placed there or because placing some other entry in cache "bumped" the desired entry out of cache.

2.2.2. Cache Management

Since the invention of cache in 1966 [Lip68], most cache management schemes can be characterized as [Smi82] [Smi87]:

- Cache is managed purely by hardware.
- The reference sequence/pattern of each particular program is not considered. Probabilistic or runtime history-based predictions of future behavior are used. The concept of live range analysis is completely ignored.
- Management of cache is transparent to programmers and compilers[4].
- All references are through the cache; if the required information is not in cache (i.e., a cache miss), the corresponding line is fetched into the cache so that the reference may be made. Of course, this implies that in normal operation every cache miss will cause a line from cache to be replaced.
- Cache can be used to hold both data and instructions.

4. There are some exceptions that have explicit cache control such as IBM 801 [Rad83]. The 801 includes cache control instructions, but little has been written describing how they should be used.

2.3. Summary of Differences

As noted at the beginning of this paper, there is no conceptual difference between the functions of registers and cache in the traditional memory hierarchy. Rather, they are distinguished by their physical aspects: speed, size, and addressing mode (cache is referenced by address and register by register name). However, from a compiler viewpoint, there are two fundamental conceptual differences between registers and cache:

[1] Since caches are accessed associatively by main memory addresses, **ambiguous alias references** — pointer or subscript operations which result in the same memory address being referenced by two or more different names (aliases) — will still reference the same item in cache; this is not true of registers. An aliased value placed in a register will have to be spilled whenever any of its possible aliases is stored into; this spilling makes registers virtually worthless for aliased values.

[2] Since most computers do not have an "execute register" instruction, there is no benefit in placing an instruction in a register.

In summary, registers can be managed more efficiently at compile-time, but cache is far more general in its application. The ideal is therefore to use registers where they are more efficient, and to use cache *only* for those tasks which cannot benefit from register use. To accomplish this, it will be necessary to slightly modify the cache so that it may be partially controlled by the compiler — this simple modification is discussed in the next section.

3. A Unified View of Cache and Registers

In the previous section we have characterized the differences between registers and cache as primarily differences in the types of items which can be profitably kept in each. Since register allocation *is* a viable compile-time management scheme and cache has none, this section will attempt to show that the concepts upon which register allocation is based are equally applicable to cache.

3.1. Live Range of References in Cache

A major concept in register allocation is that of liveness. In the literature on caches, one finds liveness mentioned only as a technique for reducing overhead in analysis of program traces [McD88]. It is also tempting to think of liveness of *addresses* — rather than liveness of values stored in them. Instead of using address or name, the definition of live range of an item to be cached should be in terms of values — exactly as in register allocation:

Definition 1: Live Range of a Value

> The live range of a value v is defined as the set of instructions during which the value v exists and may be referenced. In other words, it is the *D-U chain* of $v \cup$ all instructions which, on some flow path, may be executed after v's *def* and before the last *use* of v on that flow path.

However, unlike registers, cache may hold instructions. For this reason, it is necessary to define the live range of an instruction:

Definition 2: Live Range of an Instruction

> The live range of an instruction α is defined as the set of instructions, including α, which may be executed after the first execution of α and before the last execution of α on some flow path. Notice that for straight-line code the live range of an instruction α is always the set $\{\alpha\}$, however, if α is enclosed in a loop or multiple-caller subprogram the set may be greatly enlarged.

Although these definitions are not surprising, they do have some surprising implications.

Perhaps the most dramatic of these is that a value which has become dead need not be stored back to main memory. Hence, suppose that the compiler is able to determine that a particular memory read operation of the value v will be the last *use* of v. If the value v is cached, even if the cached value v does not match the value which is stored at the corresponding address in main memory[5], the cache cell containing value v need not be stored back to main memory. The compiler may simply inform the cache that this was the last reference to v and hence that the cache line which held v is "empty" at completion of this *use* of v[6].

---

5. This occurs if a *def* of v is executed when v is in cache or when a *def* of v creates the cache line and the cache has not yet stored v back to main memory.

6. In the interest of simplicity, this discussion has pretended that a cache line holds exactly one value — this restriction is easily removed, although more complex bookkeeping is required.

The benefit of having such "empty" cache lines is that *instead of the basic line-replace operation, only a simple placement is required to install a new line in cache.* Of course, this also insures that no useful item was accidentally flushed from cache in this process.

The only hardware modification needed to support application of this new concept is that the compiler needs to be able to tag references with a "last reference" bit. This may be accomplished in many different ways, several of which are briefly discussed in [ChD89].

3.2. Selective Caching (Cache Bypass)

In register allocation, the compiler completely determines which register will hold each value and for how long. However, a conventional cache does not provide such compile-time control; a conventional cache simply employs a program-independent hardware-implemented strategy for placing each referenced item somewhere in cache and for choosing the item to be spilled to main memory if the cache was already full.

It is found that substantial benefit can be gained by simply using a single bit of control — to determine whether the current reference should be placed in cache or if it should instead bypass the cache [ChD89].

In this paper, we assume that the compiler at least has the ability to tag each reference as to whether the reference should be made using the cache or bypassing the cache. Bypassing the cache would be done for two separate reasons:

[1] As discussed in [ChD89], some items are not referenced often enough to be worth keeping in cache. Due to the expense of loading the cache with an entire line, as compared to the cost of reading a single word, it is common that individual references would run slower with cache than without it — bypass avoids this worst-case behavior.

[2] If the benefit from keeping a particular value in a register is greater than that of keeping it in cache and a register is available, then the cache should be bypassed when that register is loaded or stored. Failure to do this wastes cache cells by making them hold values which will not be referenced, as discussed earlier.

Combining cache bypass with an underlying hardware-implemented policy provides a reasonable approximation to the full allocation control available with registers.

The least recently-used (LRU) scheme for cache line replacement chooses for replacement that line in cache which has not been referenced for the longest period of time [Spi76]. Since it may become difficult to maintain the LRU stack per se, an LRU approximation is often implemented as a one-bit time stamp indicating whether each line has been referenced since the last time the stamp was read and reset.

Suppose that a line λ is in cache and that the processor is about to make what is known to be the last reference to λ. In either LRU or an LRU approximation, the line λ would be present in cache for $O(n)$ time units after the last reference, where n is the number of lines in a *cache associative set*, because it will take that long for λ to be nudged into the least-recently-referenced position. In effect, if the average cacheable item is referenced τ times, then approximately $1/\tau$ of the cache cells will be wasted. Notice that τ=1 items would be a complete waste — something referenced only once should never be placed in cache. Further note that, in typical programs, relatively few items are referenced more than a few times (except perhaps in some loops).

To avoid this problem, we propose that the compiler mark the last reference to each item as such, and that the cache hardware would immediately interpret this as making the cache cell holding λ "empty" or, alternatively, making it the least recently used. If the line λ was not in cache, the hardware should simply make the reference to λ using the cache bypass or, alternatively, bringing the line into cache but immediately marking it as the least recently used.

If other underlying hardware-implemented replacement schemes are used, they should be modified in the same way described above for LRU. This can be done easily for FIFO, random, and even Belady's MIN algorithm [Bel66].

4. The Unified Registers/Cache Management Model

The previous section outlines the fundamental concepts for treating cache as a register-like compiler-managed entity. In this section, the complete strategy for the unified registers/cache management model is described.

First, compiler technology needed for this unified registers/cache management model is given. The management scheme of registers and cache using the unified model is then presented. Finally, the implementation techniques — both hardware and semantics — are proposed.

4.1. Compiler Support

The basic compiler technology needed for the unified registers/cache management model is very similar to that needed to perform register allocation, however, there is some complications. The first complication is that names must be grouped according to which other names they are ambiguously aliased with, henceforth called an **alias set**;

4.1.1. Alias Sets

As discussed above, the fundamental flaw in static analysis of conventional-language programs is that it is not possible to statically determine, for all variables, which ones *are* aliased to which others at each point in the program. The alias problem is simply finding which items *can be* aliased to each other. We call this problem the construction of **alias sets**.

The basic tools with which alias sets are constructed are the familiar algorithms of compiler flow analysis (including dependence analysis). These tools have been particularly well-honed in pursuit of efficient automatic parallelization. The presentation here is intended merely to provide a brief overview to the analysis involved in creating alias sets.

4.1.1.1. Names

The first issue to resolve in grouping names into alias sets is the basic question of what constitutes a name. Each variable could be considered a name, however, this is not the most useful definition. The difficulty is rooted in the fact that a variable α may be an alias for a variable β within one region of a program, while α may be an alias for δ in another section of the code. In such a case, considering α to be a name used for grouping into alias sets, it would be necessary either to make the alias set containing α be $\{\alpha,\beta,\delta\}$ or to make the alias set for α be $\{\alpha,\beta\}$ in one region of code and $\{\alpha,\delta\}$ in another. Ideally, names should be chosen so that each name is a member of an alias set whose contents are independent of position in the program, yet where no names are included unnecessarily.

The solution to this naming problem is simply to incorporate control and data flow information in the names: however, the mapping from user variable names into these **aliased-object names** is surprisingly complex. For example, if the user has declared *i* to be an *int* variable and *p* to be an *int* * which is initially set to point at *i* (e.g., *p=(&i);*), then references to both *i* and **p* use the same aliased-object name: user names are mapped many-to-one into aliased-object names. This means that if the compiler can detect that two user names are unambiguously aliased to each other, these two user names will share a single aliased-object name. The rule is more precisely expressed as:

Definition 1: User-Name Merging

> The user-created names α and β can be merged into a single aliased-object name within some region of code *iff* the values associated with the names α and β are known to be the same throughout that region of code.

which also implies that explicitly made copies of values can all share a single aliased-object name (i.e., the compiler can perform copy propagation).

On the other hand, in a code sequence like *i=j; . . . i=k;*, the user name *i* will be mapped into multiple aliased-object names, one for each different value stored into *i*. This rule is best expressed in terms of **D-U chains** and **U-D chains** [AhS86]:

Definition 2: User-Name Splitting

> Let U be the set of uses of (loads from) the user name α. For each use $u_i \in U$, let the U-D chain rooted at u_i be called d_i. If, for any i and j, $d_i \cap d_j \neq \emptyset$, then let $d_i = d_i \cup d_j$ and delete d_j. When no more such merger/deletions can be performed, each of the remaining sets (d_i) can be represented by a separate aliased-object name.

Notice that values which do not have programmer-assigned names, such as intermediate results within an expression, also may be assigned aliased-object names by the above rules.

4.1.1.2. Formation of Alias Sets

Given the above definitions, it is relatively easy for a compiler to generate a set of names appropriate for grouping into alias sets; but what is an alias set? There are actually several compile-time distinguishable types of aliases:

[1] A name α is a **true alias** of the name β if α is known to always be associated with the same value that is associated with β. (Notice that, if this is so, the two names may be merged by Definition 1 given above.)

[2] A name α is an **intersection alias** of the name β if α and β are known to share some elements of their values, however, perhaps not all elements. For example, if a is a *struct* containing members called b and c, then a and $a.b$ are intersection aliases. Intersection aliases occur most often in code referring to arrays.

[3] A name α is a **sometimes alias** of the name β if α is known to be a true or intersection alias for β under some circumstances at run-time, however, α is not an alias for β under other circumstances. For example, references to $a[i]$ and $a[5]$ are sometimes aliases if i could be equal to 5.

[4] A name α is an **ambiguous alias** for β if α is an intersection alias or sometimes alias for β, or if the compiler is unable to determine the relationship between α and β.

[5] A name α is **mutually exclusive** of β if α and β are *not* related by any of the above alias types. If, for all β, α is mutually exclusive of β, then α is **unambiguous**.

For the purpose of unified registers/cache management, an alias set is a set of names grouped by "closure" of the ambiguous alias relation. In other words, given a name n, the alias set for n consists of $n \cup$ (all names which are ambiguous aliases of n) $\cup$ (all names which are ambiguous aliases of those names) $\cup \ldots$. Notice that these alias sets have several useful properties:

Uniqueness

If α is a name in alias set S, then α is in no other alias set. This assignment is also independent of the region of code in which α is referenced.

Completeness

If α is a name, it is a member of some alias set; if α is mutually exclusive of all other names, then the alias set which contains α is a singleton set containing α.

4.2. Strategy for the Unified Model

In this section, the complete strategy for managing registers and cache using a coordinated scheme is proposed. The key idea of this scheme is to try keeping only one copy of information in either cache or registers. Hence, any inaccessible copy of information can be eliminated and the effectiveness of each memory level increases.

Perhaps the best summary is the diagram of Figure 4. As depicted in Figure 4, the unified registers/cache management model is fundamentally different from previous proposals in that it takes full advantage of the conceptual differences between registers and cache.

From the compiler's view, memory references in a program can be classified into three different types:

- ambiguous data values,
- unambiguous data values, and
- instructions.

To avoid any inaccessible copies of values in the local memory hardware, any placement of memory reference values should be done according to the usability of each memory level.

Registers are very restricted in their usability, and the conventional management schemes for registers such as graph coloring techniques [ChA81] [Cha82] [Cho83] do quite well in this domain — unambiguous value references. Here, we propose that the conventional management techniques be used, but with three differences:

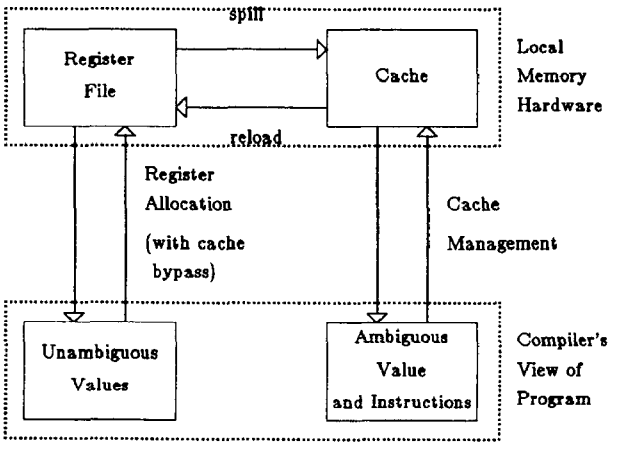

Figure 4: Unified Registers/Cache Management Model

[1] When a register will be used for a series of operations, the loading and storing of the

value into a register should bypass the cache.

[2] When a register's value must be spilled due to a shortage of registers, it should be spilled to cache.

[3] When the spilled value is referenced, it is either reloaded from:

- Cache
  In this case, the cached copy becomes dead as soon as the value is reloaded into a register.
- Main Memory
  In this case, the cache is bypassed and the value is directly referenced from the main memory.

However, cache, which is normally managed completely by hardware-implemented schemes, requires some degree of compiler control in order to achieve better system performance. Cache will be used only for register spills (see above), ambiguously named values, and for instructions. Further, cache will only be used *when it may improve performance* — rather than being used blindly for each reference.

The subdivision of references into ambiguous values, unambiguous values, and instructions is relatively straightforward compiler technology [AhS86] [Die87]. Hence, determining which references should be handled by register allocation and which by cache management is a simple matter. Since register spills should go to cache, however, there is a natural ordering that register allocation should precede cache management decisions.

4.3. Semantics for the Unified Model

The following semantics is defined for register-register operation architecture. However, they can easily be extended to other types of architectures with slightly modification.

With the unified registers/cache management model, there are four different types of load/store instructions corresponding to the fetching and storing of values in cache and registers. They are:

- Am_LOAD,
- AmSp_STORE,
- UmAm_LOAD, and
- UmAm_STORE.

A cache bypass bit per each memory reference is also used to indicate if the reference goes through the cache. A "1" would means "bypass" and a "0" means "go through the cache".

The operations of these four load/store instructions are as follows:

- **Am_LOAD**
  This type of load instructions fetches a datum into register through cache. That is, a copy of the datum will appear in cache after the reference and the cache bypass bit is set to zero. This instruction is used for loading ambiguous values.
- **AmSp_STORE**
  This type of store instructions saves a datum through cache. That is, the datum is placed in cache and the cache bypass bit is set to zero. There are two situations which use this store instruction:
  - when an ambiguous value is stored.
  - when an unambiguous value is spilled from a register.
- **UmAm_LOAD**
  The operation of this instruction is to check if the datum is cache. If it is in cache, the datum is loaded into a register and that datum in cache is then marked as invalid or empty. If it is not in cache, the datum is loaded from main memory to cache directly, bypassing the cache. In both case, the cache bypass bit is set to one. This type of load instruction is used for loading unambiguous values.
- **UmAm_STORE**
  This type of store instruction saves a datum directly to main memory, bypassing the cache. The cache bypass bit is set to one. It is used for saving unambiguous values into main memory which are not due to register spilling.

4.4. Hardware Implementation

With the results of compiler analysis of a program, the question of ambiguity of references can easily be answered so as to allow the unified model to manage registers/cache together. However, this information must be transmitted to the cache bypass control logic for each memory reference in load/store instructions. The information for each load/store instruction requires only a single bit — a 1 means "bypass" and 0 means "go through the cache." The natural question is how does the compiler get this one bit of information for each reference into the cache bypass control at

runtime?

There are a number of alternative solutions to this problem and each of these solutions trades off some resources or capabilities.

The conceptually easiest and most efficient way to transmit this cache bypass information is to embed a bit in each instruction for each memory reference the instruction may cause. For new machine design, this is fairly convenient; reserving a control bit to obtain speedups of total memory access time by factors of 2 or more is virtually always worthwhile. Also, existing machines with at least one currently unused bit in each instruction should probably use this implementation.

Alternatively, the instruction set of the machine can be expanded to include explicit cache bypass control instructions. In fact, these instructions exist for virtually all computers which have cache. An extreme example of this explicit cache control is the IBM 801, where individual cache lines can be explicitly allocated and deallocated; most systems simply permit the cache to be enabled/disabled as a whole. Since bypasses may come in "clumps", even this crude bypass control can gain some improvement; however, bypasses do not always come in clumps. By defining a new instruction specifically to implement cache bypass control, one could permit each cache control instruction to set the pattern of bypass/cache decisions for the next n references, where n is somewhat less than the machine word length. Again, some performance would be gained, but the high frequency of cache bypass control instructions would limit performance.

While all the above schemes have some merit, there is another scheme which both permits a cache control bit to be associated with each instruction and does not require changes in the instruction set design or encoding. In current machine designs, the addressable space is typically very large and programs rarely use the entire addressable space of the machine. Thus, it is possible to trade one address bit (e.g., the most significant bit of an address) for use as the control bit for the cache bypass. In fact, this solution is suggested by Intel in their 80386 programmer's reference manual [Int86] as a way to provide a cache control bit for use in multiprocessor cache coherency control. Worst case, this effectively reduces the addressable space by 50%[7]. Of course, it also causes the compiler writer a bit of grief in that not only must all addresses be correctly tagged, but the compiler must also be careful about operations such as pointer arithmetic or comparisons.

Other methods, such as using a separate cache controller to explicitly control the cache (similar to the remote PC idea [Rad83]) are also possible. However, the overhead and the synchronization cost involved may be too large to be practical.

5. Simulation Result

To measure the effect of the unified management scheme in reducing memory traffic in data cache, simulation study of the MIPS architecture to measure ambiguous and unambiguous data value references was performed. The benchmark programs were taken from the DARPA MIPS package, and are widely used as benchmarks of cache and register performance. Data are given for these programs:

- Bubble — a typical bubble sort program, executed on a set of 500 random data.
- Intmm — a program which performs a matrix multiplication of two integer matrices, each of which is 40 by 40.
- Puzzle — a compute-bound program from Forest Basket, which runs with a size of 511.
- Queen — a program to solve the 8 queens problem.
- Sieve — a program to calculate the number of primes between 0 and 8190.
- Towers — the standard recursive tower-of-Hanoi solution, given the problem of moving 18 disks.

The result is summarized in Figure 5. Statically, about 70 to 80 percent of the load/stored data references might be marked as unambiguous and should be bypassed the cache. Runtime measurement showed that about 45 to 75 percent of the loaded/stored data references are unambiguous. Hence, memory traffic in data cache might possibly be reduced by about 60 percent by using the unified memory management scheme.

7. The actual address space may not be affected because address mapping mechanisms may be able to circumvent the loss.

Benchmark	Static Count	Dynamic Count
bubble	81.8%	45.5%
intmm	89.9%	67.8%
puzzle	80.9%	64.2%
queen	71.8%	76.1%
sieve	72.7%	23.2%
towers	80.2%	78.5%

Figure 5: Percent of Data Cache Reference Traffic Reduction

6. Conclusion

Registers and cache are *not* interchangeable, but are complementary to each other. A machine with 1000000 registers would not be able to place all values in registers, because registers cannot resolve ambiguously aliased references. A machine with 1000000 words of cache but no registers could, however, be equally futile in that without the compile-time management associated with registers there is no provision for avoiding worst-case cache scenarios where the machine would spend more time placing lines in cache and referencing them there than it would spend performing references directly from main memory (faster without cache than with it); even discounting that effect, cache access time is nearly always longer than register access time so using cache where registers would suffice is not optimal.

Miller found that the ratio of unambiguous references to ambiguous references, measured statically, is from 1:1 to 3:1 [Mil88]. This does suggest that registers are more important than cache, however, it does not count instruction references. Hence, the load placed on each type of memory is considerable.

Given these surprising realizations, we have proposed a coordinated registers/cache management scheme which can use each hardware structure for the references for which it is best suited. This technique is both implementable and familiar — very closely related to register allocation.

References

[AgC87] Agarwal, A., Chow, P., Horowitz, M., Acken, J., Salz, A., Hennessy, J., "On-Chip Instruction Caches for High Performance Processors," *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, edited by Losleben, P., The MIT Press, 1987, pp. 1-24.

[AhS86] Aho, A. V., Sethi, R., Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts, 1986.

[All83] Allen, J.R., *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*, Rice University, Ph.D. Thesis, April 1983.

[All86] Allen, F., "The Parallel Translator Project," *NASA / ICASE Parallel Languages and Environments Workshop*, November 1986.

[Bel66] Belady, L.A., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM System Journal*, Volume 5, 1966, pp. 78-101.

[BuC86] Burke, M., Cytron, R., "Interprocedural Dependence Analysis and Parallelization," *Proceeding of the SIGPLAN Symposium on Compiler Construction, 1986*, pp. 162-175.

[Bur84] M. Burke, "An Interval Analysis Approach Toward Interprocedural Data Flow," *Research Report RC 10640 (#47724)*, IBM, Yorktown Heights, New York, July 1984.

[ChA81] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W., "Register Allocation Via Coloring," *Computer Languages*, Volume 6, 1981, pp. 47-57.

[Cha82] Chaitin, G.J., "Register Allocation and Spilling via Graph Coloring," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notice* Volume 17, Number 6, June 1982, pp. 201-207.

[ChD89] Chi, C.H., Dietz, H., "Improving Cache Performance by Selective Cache Bypass," *Proceeding of the 1988 Hawaii International Conference on Systems Sciences*, January 1989, pp. 256-265.

[ChH84] Chow, F., Hennessy, J., "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,*

SIGPLAN Notice Volume 19, Number 6, June 1984, pp. 222-232.

[Cho83] Chow, F.C., "A Portable Machine-Independent Global Optimizer-Design and Measurement," *Technical Note no. 83-254*, Computer Systems Laboratory, Stanford University, December 1983.

[Die87] Dietz, H.G., "The Refined-Language Approach to Compiling for Parallel Supercomputers," *Ph.D. Thesis*, Polytechnic University, June 1987.

[Fre74] Freiburghouse, R.A., "Register Allocation Via Usage Courts," *Communications of the ACM*, Volume 17, Number 11, November 1974, pp. 638-642.

[HeJ83] Hennessy, J., Jouppi, N., Baskett, F., Gill, J., "MIPS: A VLSI Processor Architecture," *Technical Report No. 223, Computer Systems Laboratory*, Stanford University, June 1983.

[Hil83] Hill, M.D., "Evaluation of On-Chip Cache," *M.S. Thesis*, University of California, Berkeley, December, 1983.

[Int86] Intel Corporation, *80386 programmer's reference manual*, 1986, pp. 11-6.

[KaM87] Kadota, H., Miyake, J., Okabayashi, I., Maeda, T., Okamota, T., Nakajima, M., Kagawa, K., "A 32-bit CMOS Microprocessor with On-Chip Cache and TLB," *IEEE Journal of Solid-State Circuits*, Volume SC-32, Number 5, October 1987, pp. 800-807.

[Kat83] Katevenis, M.H., "Reduced Instruction Set Computer Architectures for VLSI," *Ph.D Thesis*, Department of Electrical and Computer Science, University of California, Berkeley, 1983.

[Lee87] Lee, R.L., "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors," *Ph.D Thesis*, University of Illinois at Urbana-Champaign, May 1987.

[Lip68] Liptay, J., "Structural Aspects of the System/360 Model 85: Part II: The Cache," *IBM Systems Journal*, 1968, pp. 15-21.

[McD88] McNiven, G.D., Davidson, E.S., "Analysis of Memory Referencing Behavior for Design of Local Memories," *Proc. of the 15th Annual Int'l. Symposium on Computer Architecture*, June 1988, pp. 56-63.

[Mil88] Miller, B.P., "The Frequency of Dynamic Pointer References in "C" Programs," *Computer Sciences Technical Report #759*, University of Wisconsin, Madison, March 1988.

[Pat85] Patterson, D.A., "Reduced Instruction Set Computers," *Communications of the ACM*, Volume 28, Number 1, January 1985, pp. 8-21.

[Rad83] Radin, G., "The 801 Minicomputer," *IBM Journal of Research and Development*, May 1983, pp. 237-246.

[Smi82] Smith, A.J., "Cache Memories," *Computing Surveys*, Volume 14, Number 3, September, 1982, pp. 473-530.

[Smi87] Smith, A.J., "Cache Memory Design: An Evolving Art," *IEEE Spectrum*, December 1987, pp. 40-44.

[Spi76] Spirn, J., "Distance String Models for Program Behavior," *IEEE Computer*, November, 1976, pp. 14-20.

[Ste86] Stein, K., "Refined C Compiler Status Report", Internal Report, Stevens Institute of Technology, 1986.